conference
proceedings

14th USENIX Symposium on Networked Systems Design and Implementation

Boston, MA, USA
March 27–29, 2017

Sponsored by
usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION
In cooperation with ACM SIGCOMM and ACM SIGOPS

14th USENIX Symposium on Networked Systems Design and Implementation
Boston, MA, USA March 27–29, 2017

ISBN 978-1-931971-37-9

Thanks to Our NSDI '17 Sponsors

Gold Sponsors


Silver Sponsors


Bronze Sponsors



Media Sponsors and Industry Partners

ACM Queue No Starch Press O'Reilly Media

ISBN 978-1-931971-37-9

Thanks to Our USENIX Supporters

USENIX Patrons

Facebook Google Microsoft NetApp

USENIX Benefactor

VMware

Open Access Publishing Partner

PeerJ

Proceedings of NSDI '17: 14th USENIX Symposium on Networked Systems Design and Implementation

March 27–29, 2017
Boston, MA

Symposium Organizers

Program Co-Chairs

Aditya Akella, *University of Wisconsin–Madison*
Jon Howell, *Google*

Program Committee

Sharad Agarwal, *Microsoft*
Tom Anderson, *University of Washington*
Andrea Arpaci-Dusseau, *University of Wisconsin—Madison*
Anirudh Badam, *Microsoft*
Mahesh Balakrishnan, *Yale University*
Fabian Bustamante, *Northwestern University*
Ranveer Chandra, *Microsoft*
David Choffnes, *Northeastern University*
Romit Roy Choudhury, *University of Illinois at Urbana–Champaign*
Mosharaf Chowdhury, *University of Michigan*
Mike Dahlin, *Google*
Anja Feldmann, *Technische Universität Berlin*
Rodrigo Fonseca, *Brown University*
Nate Foster, *Cornell University*
Deepak Ganesan, *University of Massachusetts Amherst*
Phillipa Gill, *University of Massachusetts Amherst*
Srikanth Kandula, *Microsoft*
Teemu Koponen, *Styra*
Swarun Kumar, *Carnegie Mellon University*
Sanjeev Kumar, *Uber*
Jacob Lorch, *Microsoft*
Dahlia Malkhi, *VMware*
Dave Maltz, *Microsoft*
Z. Morley Mao, *University of Michigan*
Michael Mitzenmacher, *Harvard University*
Jason Nieh, *Columbia University*
George Porter, *University of California, San Diego*
Luigi Rizzo, *University of Pisa*
Srini Seshan, *Carnegie Mellon University*
Anees Shaikh, *Google*
Ankit Singla, *ETH Zurich*
Robbert van Renesse, *Cornell University*
Geoff Voelker, *University of California, San Diego*
David Wetherall, *Google*
Adam Wierman, *California Institute of Technology*
John Wilkes, *Google*
Minlan Yu, *University of Southern California*
Heather Zheng, *University of California, Santa Barbara*
Lin Zhong, *Rice University*

Poster Session Co-Chairs

Anirudh Badam, *Microsoft*
Mosharaf Chowdhury, *University of Michigan*

Steering Committee

Katerina Argyraki, *EPFL*
Paul Barham, *Google*
Nick Feamster, *Georgia Institute of Technology*
Casey Henderson, *USENIX Association*
Jeff Mogul, *Google*
Brian Noble, *University of Michigan*
Timothy Roscoe, *ETH Zurich*
Alex C. Snoeren, *University of California, San Diego*

External Reviewers

Raluca Ada
Omid Alipourfard
Andrew Baumann
David Bindel
Eleanor Birrell
Matt Burke
Nishanth Chandran
Melissa Chase
Robert Escriva
Ittay Eyal
Jiaqi Gao
Aaron Gember-Jacobson
Efe Gencer
Monia Ghobadi
Joseph Gonzalez
Chris Hodsdon
Robert D. Kleinberg
Yuliang Li
Haonan Lu
Rui Miao
Shuai Mu
Kartik Nayak
Khiem Ngo
Amar Phanishayee
Ling Ren
Vyas Sekar
Kevin Sekniqi
Rob Shakir
Isaac Sheff
Zhiming Shen
Yoram Singer
Theano Stavrinos
Michael Swift
Edward Tremel
Nickolai Zeldovich
Xinyu Zhang

NSDI '17: 14th USENIX Symposium on Networked Systems Design and Implementation
March 27–29, 2017
Boston, MA

Monday, March 27, 2017

Storage Systems

Packet Processing

Security and Privacy

(Continues on next page)

(Continues on next page)

Data-Driven Systems

Datacenter Networking

Cloud and Distributed Systems

Wednesday, March 29

Mobile Systems and IoT

(Continues on next page)

Networking in the Datacenter

Big Data Systems

Network Verification and Debugging

Message from the NSDI '17 Program Co-Chairs

Welcome to NSDI '17! We are delighted to continue the NSDI tradition and share with you the latest research on networked systems. The NSDI '17 program showcases new and exciting advances in storage systems, packet processing, security and privacy, big data analytics, and data center networking. This year's Operational Systems Track—which describes experience with real, deployed networks—features work on virtual switch platforms for the public cloud and petabyte-scale storage compression.

NSDI '17 received 254 submissions of which we accepted 46 papers. Our Program Committee consisted of 42 members with a mix of research and industry experience. NSDI '17 adopted double-blind reviewing. The committee rejected 12 papers without reviews for formatting violations. The remaining papers were reviewed in two rounds: in the first round, each paper received three reviews; the 110 papers that advanced to the second round received at least three more reviews. Once we completed reviewing, the committee discussed online and selected 75 papers that were discussed further at a 1.5-day PC meeting held in Seattle, WA. The program committee strived to produce valuable feedback; we hope it benefited authors of every submission.

It has been a great pleasure working with many other people to put this program together. We would like to thank the authors of all submitted papers for choosing to send work of such high caliber to NSDI. Thanks also to the program committee for their professionalism, diligence and enthusiasm. Special thanks to Anirudh Badam and Mosharaf Chowdhury for serving as poster chairs and to Anirudh Badam, Jacob Lorch, and Adam Wierman for serving on the Awards Committee. We are also very grateful to the USENIX staff, including Casey, Hilary and Michele, for their exceptional support and help. Finally, NSDI wouldn't be what it is without the attendees, so thank you very much for being here. We hope you enjoy the conference!

Aditya Akella, *University of Wisconsin–Madison*
Jon Howell, *Google*
NSDI '17 Program Co-Chairs

The Design, Implementation, and Deployment of a System to Transparently Compress Hundreds of Petabytes of Image Files For a File-Storage Service

Daniel Reiter Horn
Dropbox

Ken Elkabany
Dropbox

Chris Lesniewski-Laas
Dropbox

Keith Winstein
Stanford University

Abstract

We report the design, implementation, and deployment of Lepton, a fault-tolerant system that losslessly compresses JPEG images to 77% of their original size on average. Lepton replaces the lowest layer of baseline JPEG compression—a Huffman code—with a parallelized arithmetic code, so that the exact bytes of the original JPEG file can be recovered quickly. Lepton matches the compression efficiency of the best prior work, while decoding more than nine times faster and in a streaming manner. Lepton has been released as open-source software and has been deployed for a year on the Dropbox file-storage backend. As of February 2017, it had compressed more than 203 PiB of user JPEG files, saving more than 46 PiB.

1 Introduction

In the last decade, centrally hosted network filesystems have grown to serve hundreds of millions of users. These services include Amazon Cloud Drive, Box, Dropbox, Google Drive, Microsoft OneDrive, and SugarSync.

Commercially, these systems typically offer users a storage quota in exchange for a flat monthly fee, or no fee at all. Meanwhile, the cost to operate such a system increases with the amount of data actually stored. Therefore, operators benefit from anything that reduces the net amount of data they store.

These filesystems have become gargantuan. After less than ten years in operation, the Dropbox system contains roughly one exabyte (± 50%) of data, even after applying techniques such as deduplication and zlib compression.

We report on our experience with a different technique: format-specific transparent file compression, based on a statistical model tuned to perform well on a large corpus.

In operating Dropbox, we observed that JPEG images [10] make up roughly 35% of the bytes stored. We suspected that most of these images were compressed inefficiently, either because they were limited to "baseline" methods that were royalty-free in the 1990s, or because they were encoded by fixed-function compression chips.

In response, we built a compression tool, called Lepton, that replaces the lowest layer of baseline JPEG images—the lossless Huffman coding—with a custom statistical model that we tuned to perform well across a broad corpus of JPEG images stored in Dropbox. Lepton losslessly compresses an average JPEG image by about 23%. This is expected to save Dropbox more than 8% of its overall backend file storage. Lepton introduces new techniques that allow it to match the compression savings of prior work (PackJPG) while decoding more than nine times faster and in a streaming manner (Figure 1).

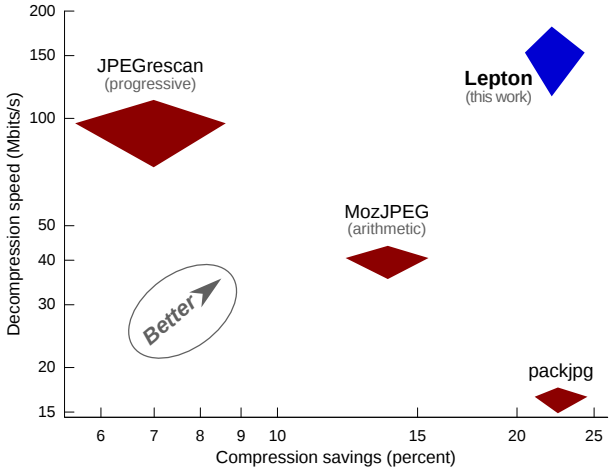


Figure 1: Compression savings and decompression speed (time-to-last-byte) of four lossless JPEG compression tools. Diamonds show 25th, 50th, and 75th percentiles across 200,000 JPEGs between 100 KiB and 4 MiB (§ 4).

Some of the challenges in building Lepton included:

- **Round-trip transparency.** Lepton needs to deterministically recover the exact bytes of the original file, even for intentionally malformed files, and even after updates to the Lepton software since the file was originally compressed.
- **Distribution across independent chunks.** The Dropbox back-end stores files in independent 4-MiB chunks across many servers. Lepton must be able to decompress any substring of a JPEG file, without access to other substrings.
- **Low latency and streaming.** Because users are sensitive to file download latency, we must optimize decompression—from Lepton's format back into Huffman-coded JPEG output—for both time-to-first-byte and time-to-last-byte. To achieve this, the Lepton format includes "Huffman handover words" that enable the decoder to be multithreaded and to start transmitting bytes soon after a request.

- **Security.** Before reading input data, Lepton enters a restricted environment where the only allowed system calls are `read`, `write`, `exit`, and `sigreturn`. Lepton must pre-spawn threads and allocate all memory before it sees any input.
- **Memory.** To preserve server resources, Lepton must work row-by-row on a JPEG file, instead of decoding the entire file into RAM.

We deployed Lepton in April 2016 on the production Dropbox service. Compression is applied immediately to all uploads of new JPEG files. We are gradually compressing files in existing storage and have ramped up to a rate of roughly a petabyte per day of input, consuming about 300 kilowatts continuously. As of February 2017, Lepton had been run on more than 150 billion user files, accounting for more than 203 PiB of input. It reduced their size by a total of more than 46 PiB. We have released Lepton as open-source software [1].

We report here on Lepton's design (§ 3), evaluation (§ 4), and production deployment (§ 5), and share a number of case studies and anomalies (§ 6) encountered in operating the system at a large scale.

2 Related Work

In network filesystems and storage services, four general categories of approaches are used to compress user files.

Generic entropy compression. Many filesystems compress files using generic techniques, such as the Deflate algorithm [6], which combines LZ77 compression [23] and Huffman coding [8] and is used by software such as zlib, pkzip, and gzip. More recent algorithms such as LZMA [15], Brotli [3], and Zstandard [5] achieve a range of tradeoffs of compression savings and speed.

In practice, none of these algorithms achieve much compression on "already compressed" files, including JPEGs. In our evaluation on 200,000 randomly selected user JPEG images, each of these algorithms achieved savings of 1% or less (§ 4).

Lossy image compression. A second category of approach involves decoding user images and re-compressing them into a more-efficient format, at the cost of some visual fidelity. The "more-efficient format" may simply be a lower-resolution or lower-quality JPEG. The JPEG files produced by fixed-function hardware encoders in cellular-phone cameras can typically be reduced in size by 70% before users see a perceptual difference [19]. The re-compression may also use more sophisticated techniques such as WebP or single-frame H.265. With this approach, storage savings for the provider can be as high as users are willing to tolerate. However, Dropbox does not modify user files, precluding these approaches.

Format-aware pixel-exact recompression. Several tools let users re-compress JPEG files to achieve more-efficient compression without affecting the decoded image. These tools include JPEGrescan [13] and MozJPEG [2], both based on the jpegtran tool written by the Independent JPEG Group [9]. The tools employ two key techniques: replacing Huffman coding with arithmetic coding (which is more efficient but was patent-encumbered when JPEG was formalized and is not part of the baseline JPEG standard), and rewriting the file in "progressive" order, which can group similar values together and result in more efficient coding. Xu et al. developed a related algorithm which uses a large number of context-dependent Huffman tables to encode JPEG images, reporting average compression savings of 15% in 30-50ms [22]. These tools preserve the exact pixels of the decoded image, but do not allow bit-exact round-trip recovery of the original file.

Format-aware, file-preserving recompression. A final set of tools can re-compress a JPEG file with round-trip recovery of the exact bytes of the original file. These include PackJPG [17, 11] and PAQ8PX [12]. These tools use different techniques, but in our evaluations, achieved roughly the same compression savings (23%) on average.

These tools use techniques unavailable in a real-time setting. For example, one of PackJPG's compression techniques requires re-arranging all of the compressed pixel values in the file in a globally sorted order. This means that decompression is single-threaded, requires access to the entire file, and requires decoding the image into RAM before any byte can be output. The time-to-first-byte and time-to-last-byte are too high to satisfy the service goals for the Dropbox service. PAQ8PX is considerably slower.

Lepton was inspired by PackJPG and the algorithms developed by Lakhani [11], and Lepton uses the same JPEG-parsing routines that PackJPG uses (uncmpjpg). However, unlike PackJPG, Lepton only utilizes compression techniques that can be implemented without "global" operations, so that decoding can be distributed across independent chunks and multithreaded within each chunk.

3 Lepton: Design and Implementation

At its core, Lepton is a stand-alone tool that performs round-trip compression and decompression of baseline JPEG files. We have released Lepton as open-source software [1]; it builds and runs on Linux, MacOS, Windows, iOS, Android, and Emscripten (JavaScript).

In its current deployment at Dropbox, Lepton is executed directly by a back-end file server or, when a file server is under high load, execution is "outsourced" from the file server to a cluster of machines dedicated to Lepton only (§ 5.5). Thus, compression and decompression is currently transparent to client software and does not reduce network utilization. In the future, we may include Lepton in the client software, in order to save network bandwidth and distribute the computation load.

In contrast to related work (§ 2), Lepton was designed to meet constraints specific to real-time compression and decompression in a distributed network filesystem:

Distribution across independent chunks. Decompression must be able to be distributed across independent pieces. The Dropbox back-end stores files in chunks of at most 4 MiB, spread across many servers. Client software retrieves each chunk independently. Therefore, Lepton must be able to decompress any substring of a JPEG file, without access to other substrings. By contrast, the *compression* process is not subject to the same constraints, because performance does not affect user-visible latency. In practice, Lepton compresses the first chunk in a file immediately when uploaded, and compresses subsequent chunks later after assembling the whole file in one place.

Within chunks, parallel decoding and streaming. Decoding occurs when a client asks to retrieve a chunk from a network file server, typically over a consumer Internet connection. Therefore, it is not sufficient for Lepton simply to have a reasonable time-to-last-byte for decompressing a 4-MiB chunk. The file server must start streaming bytes quickly to start filling up the client's network connection, even if the whole chunk has not yet been decompressed.

In addition, average decoding speed must be fast enough to saturate a typical user's Internet connection ($>$ 100 Mbps). In practice, this means that decoding must be multithreaded, including both the decoding of the Lepton-format compressed file (arithmetic code decoding) and the re-encoding of the user's Huffman-coded baseline-JPEG file. To accomplish the latter, the Lepton-format files are partitioned into segments (one for each decoding thread), and each thread's segment starts with a "Huffman handover word" to allow that thread's Huffman encoder to resume in mid-symbol at a byte boundary.

We now give an overview of JPEG compression and discuss the design of Lepton subject to these requirements.

3.1 Overview of JPEG compression

A baseline JPEG image file has two sections—headers (including comments) and the image data itself (the "scan"). Lepton compresses the headers with existing lossless techniques ([6]). The "scan" encodes an array of quantized coefficients, grouped into sets of 64 coefficients known as "blocks." Each coefficient represents the amplitude of a particular 8x8 basis function; to decode the JPEG itself, these basis functions are summed, weighted by each coefficient, and the resulting image is displayed. This is known as an inverse Discrete Cosine Transform.

In a baseline JPEG file, the coefficients are written using a Huffman code [8], which allows more-probable values to consume fewer bits in the output, saving space overall. The Huffman "tables," given in the header, define the probability model that determines which coefficient values will be considered more or less probable. The more accurate the model, the smaller the resulting file.

Lepton makes two major changes to this scheme. First, it replaces the Huffman code with an arithmetic code[1], a more efficient technique that was patent-encumbered at the time the JPEG specification was published (but no longer). Second, Lepton uses a sophisticated adaptive probability model that we developed by testing on a large corpus of images in the wild. The goal of the model is to produce the most accurate predictions for each coefficient's value, and therefore the smallest file size.

3.2 Lepton's probability model: no sorting, but more complexity

Arithmetic probability models typically use an array of "statistic bins," each of which tracks the probability of a "one" vs. a "zero" bit given a particular prior context. The JPEG specification includes extensions for arithmetic-coded files [10], using a probability model with about 300 bins.[2] The PackJPG tool uses about 6,400 bins, after sorting every coefficient in the image to place correlated coefficients in the same context.

In designing Lepton, we needed to avoid global operations (such as sorting) that defeat streaming or multithreaded decoding. One key insight is that such operations can be avoided by expanding the statistical model to cover correlations across long distances in the file, without needing to sort the data. Lepton's model uses 721,564 bins, each applied in a different context.

These contexts include the type of coefficient, e.g. "DC" (the basis function that represents the average brightness or color over an 8x8 block) vs. "AC," and the index of an "AC" coefficient within a block. Each coefficient is encoded with an Exp-Golomb code [18], and statistic bins are then used to track the likelihood of a "one" bit in this encoding, given the values of already-encoded coefficients that may be correlated.

At the start of an encoding or decoding thread, the statistic bins are each initialized to a 50-50 probability of zeros vs. ones. The probabilities are then adapted as the file is decoded, with each bin counting the number of "ones" and "zeroes" encountered so far.

The bins are independent, so a "one" seen in one context will not affect the prediction made in another. As a result, the number and arrangement of bins is important: compression efficiency suffers from the curse of dimensionality if too many bins are used, because the coder/decoder cannot learn useful information from similar contexts.

[1] Lepton implements a modified version of a VP8 [4] range coder.

[2] Performance is shown in Figure 1 in the diamond labeled "MozJPEG (arithmetic)." Arithmetic-coded JPEGs are not included in the widely-supported "baseline" version of the specification because they were patent-encumbered at the time the standard was published.

3.3 Details of Lepton's probability model

We developed Lepton's probability model empirically, based on a handful of photos that we captured with popular consumer cameras. We then froze the model and tested on randomly selected images from the Dropbox filesystem; performance on the "development" images correlated well with real-world performance (§ 4). The development images and the full probability model are included in the open-source release [1] and are detailed in Appendix A.2. We briefly summarize here.

For each 8x8 JPEG block, Lepton encodes 49 AC coefficients (7x7), 14 "edge" AC coefficients of horizontal (7x1) and vertical (1x7) variation, and 1 DC coefficient.

For the 7x7 AC coefficients, we predict the Golomb code bits by averaging the corresponding coefficients in the *above*, *left*, and *above-left* blocks. Hence, the bins for bits of C_i are indexed by $\langle i, \lfloor \log_2(|A_i| + |L_i| + \frac{1}{2}|AL_i|) \rfloor \rangle$.

For the 7x1 and 1x7 AC coefficients, we use the intuition supplied by Lakhani [11] to transform an entire column of a two-dimensional DCT into a one-dimensional DCT of an edge row. In this manner we can get pixel-adjacent 1D DCT coefficients from the bottom-most row of the *above* block and the top row of the current block. Likewise, we can use the neighboring right-most column of the *left* block to predict the left-most 1D DCT column of the current block.

To predict the DC coefficient, we assume image gradients are smooth across blocks. Linearly extrapolating the last two rows of pixels of the *above* and *left* blocks yields 16 edge pixel values. Since the DC coefficient is decoded last, we can use every AC coefficient to compute a predicted DC offset which minimizes average differences between the decoded block's edge pixels and the edges extrapolated from neighbors. We only encode the delta between our predicted DC value and the true DC value, so close predictions yield small outputs. We achieved additional gains by indexing the statistics bins by outlier values and the variance of edge pixels, enabling Lepton's model to adapt to non-smooth gradients.

These techniques yield significant improvements over using the same encoding for all coefficients (§ 4.3).

3.4 Decompressing independent chunks, with multithreaded output

When a client requests a chunk from the Dropbox filesystem, the back-end file servers must run Lepton to decode the compressed chunk back into the original JPEG-format bytes. Conceptually this requires two steps: *decoding* the arithmetic-coded coefficients (using the Lepton probability model) and then *encoding* the coefficients using a Huffman code, using the Huffman probability model given in the file headers.

The first step, arithmetic decoding, can be parallelized by splitting the coefficients into independent segments, with each segment decoded by one thread. Because the Lepton file format is under our control, we can use any number of such segments. However, adding threads decreases compression savings, because each thread's model starts with 50-50 probabilities and adapts independently.

The second step, Huffman encoding, is more challenging. The user's original JPEG file is not under our control and was not designed to make multithreaded encoding possible. This step can consume a considerable amount of CPU resources in the critical path and would consume a large fraction of the total latency if not parallelized. Moreover, Dropbox's 4-MiB chunks may split the JPEG file arbitrarily, including in the middle of a Huffman-coded symbol. This presents a challenge for distributed decoding of independent chunks.

To solve this, we modified the Lepton file format to include explicit "Huffman handover words" at chunk and thread boundaries. This represents state necessary for the JPEG writer to resume in the middle of a file, including in mid-symbol. In particular, the Huffman handover words include the previous DC coefficient value (16 bits), because DC values are encoded in the JPEG specification as deltas to the previous DC value, making each chunk dependent on the previous. They also include the bit alignment or offset and partial byte to be written.

The Huffman handover words allow decoding to be parallelized both across segments within a chunk, and across chunks distributed across different file servers. Within a chunk, the Huffman handover words allow separate threads to each write their own segment of the JPEG output, which can simply be concatenated and sent to the user. The Lepton file format also includes a Huffman handover word and the original Huffman probability model at the start of each chunk, allowing chunks to be retrieved and decoded independently.

4 Performance evaluation

For our benchmarks, we collected 233,376 randomly sampled data chunks beginning with the JPEG start-of-image marker (0xFF, 0xD8) from the Dropbox store. Some of these chunks are JPEG files, some are not JPEGs, and some are the first 4 MiB of a large JPEG file. Since Lepton in production is applied on a chunk-by-chunk basis, and 85% of image storage is occupied by chunks with the JPEG start-of-image marker, this sampling gives a good approximation to the deployed system.

Lepton successfully compresses 96.4% of the sampled chunks. The remaining 3.6% of chunks (accounting for only 1.2% of bytes) were non-JPEG files, or JPEGs not supported by Lepton. Lepton detects and skips these files.

4.1 Size and speed versus other algorithms

In Figure 2 we compare Lepton's performance against other compression algorithms built with the Intel C++

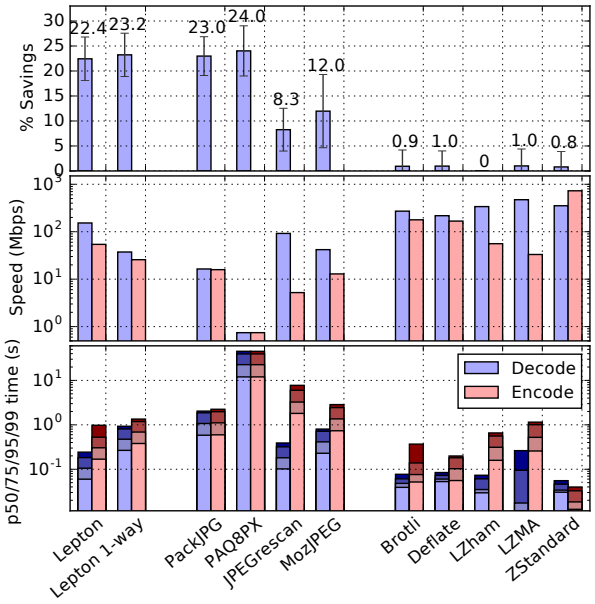


Figure 2: Compression savings and speed of codecs on the benchmark data-set (including chunks that Lepton cannot compress). Generic codecs (right) are fast, but only able to compress the JPEG header. JPEG-aware codecs (center) compress well, but are slow. Lepton (far left) is fast and compresses well.

Compiler 16.0 (icc) on a 2.6 GHz Intel Xeon E5 2650v2. We used the full benchmark data-set, including chunks that Lepton rejects as corrupt, progressive, or in the CMYK color space.

Lepton is the fastest of any format-aware compression algorithm, and it compresses about as well as the best-in-class algorithms. We also evaluated a single-threaded version of Lepton (Lepton 1-way), which we modified for maximum compression savings, by tallying statistic bins across the whole image rather than independent thread-segments. The format-aware PAQ8PX algorithm edges out single-threaded Lepton's compression ratio by 0.8 percentage points, because it incorporates a variety of alternative compression engines that work on the 3.6% of files that Lepton rejects as corrupt. However, PAQ8PX pays a price in speed: it encodes 35 times slower and decodes 50 times slower than single-threaded Lepton.

In production, we care about utilizing our CPU, network and storage resources efficiently (§ 5.6.1), and we care about response time to users. Lepton can use 16 CPU cores to decode at 300 Mbps by processing 2 images concurrently. For interactive performance, we tuned Lepton to decode image chunks in under 250 ms at the 99th percentile (p99), and the median (p50) decode time is under 60 ms. This is an order of magnitude faster than PackJPG, 1.5×–4× faster than JPEGrescan and MozJPEG, and close to Deflate or Brotli. Encoding is also fast: 1 s at the 99th percentile and 170 ms in the median case, substantially better than any other algorithm that achieves appreciable compression.

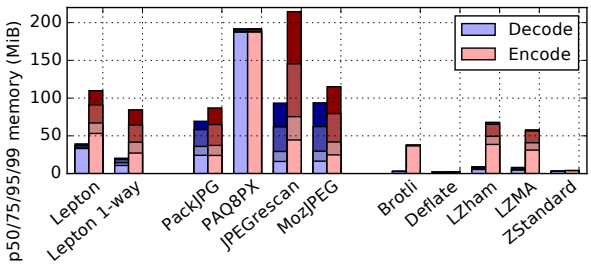


Figure 3: Max resident memory used by different algorithms.

Category	Original bytes	Compression Ratio	Bytes saved
Header	2.3% ± 4.2	47.6% ± 19.8	1.0% ± 1.8
7x7 AC	49.7% ± 7.1	80.2% ± 3.2	9.8% ± 1.7
7x1/1x7	39.8% ± 4.7	78.7% ± 3.9	8.6% ± 2.2
DC	8.2% ± 2.6	59.9% ± 8.7	3.4% ± 1.6
Total	100%	77.3% ± 3.6	22.7% ± 3.6

Figure 4: Breakdown of compression ratio (compressed size / uncompressed size) by JPEG file components.

4.2 Memory usage

Lepton shares production hosts with other, memory-hungry processes, so limiting memory usage was a significant design goal. This is particularly important for the decode path because, under memory pressure, our servers can fall back to encoding using Deflate, but we do not have that option for decoding.

Figure 3 shows the memory usage of Lepton and other algorithms on our benchmark. For decoding, single-threaded Lepton uses a hard maximum of 24 MiB to store the model and temporary buffers. Multithreaded Lepton duplicates the model for each thread, using 39 MiB at the 99th percentile. This compares favorably with other algorithms using 69–192 MiB.

Decoding can stream the output, but encoding currently retains all pixels in memory so each thread can operate on its own spatially contiguous region of pixels. Thus, the memory profile of Lepton encoding is similar to PackJPG and MozJPEG.

4.3 Compression savings by component

For the sampled chunks that Lepton successfully compresses, Figure 4 shows how each part of the file contributed to the total compression ratio, among JPEG files, of 77.3% ± 3.6 (with multithreading enabled).

Lakhani-inspired edge prediction [11] contributes 1.5% of overall compression savings. Compared with baseline PackJPG [17] (which used the same predictions for all AC coefficients), it improved the compression of 7x1/1x7 AC coefficients from 82.5% to 78.7%. DC gradient prediction contributes 1.6% of overall savings, improving the compression of DC coefficients from 79.4% (using baseline PackJPG's approach) to 59.9%. [3]

[3] By "baseline PackJPG", we refer here to the algorithm described in the 2007 publication [17]. However, for fairness, all other performance comparisons in this paper (e.g., Figures 1, 2, 3) use the latest version

5 Deployment at Scale

To deploy Lepton at large scale, without compromising durability, we faced two key design requirements: determinism and security. Our threat model includes intentionally corrupt files that seek to compress or decompress improperly or cause Lepton to execute unintended or arbitrary code or otherwise consume excess resources.

With determinism, a single successful roundtrip test guarantees that the file will be recoverable later. However, it is difficult to prove highly optimized C++ code to be either deterministic or secure, even with bounds checks enabled. Undefined behavior is a core mechanism by which C++ compilers produce efficient code [21], and inline assembly may be required to produce fast inner loops, but both hinder analysis of safety and determinism. At present, safer languages (including Rust and Java) have difficulty achieving high performance in image processing without resorting to similarly unsafe mechanisms.

We wrote Lepton in C++, and we enforce security and determinism using Linux's secure computing mode (`SECCOMP`). We have also cross-tested Lepton at scale using multiple different compilers.

5.1 Security with SECCOMP

When `SECCOMP` is activated, the kernel disallows all system calls a process may make except for `read`, `write`, `exit` and `sigreturn`. This means a program may not open new files, fork, or allocate memory. Lepton allocates a zeroed 200-MiB region of memory upfront, before reading user data, and sets up pipes to each of its threads before initiating `SECCOMP`. Memory is allocated from the main thread to avoid the need for thread synchronization.

5.2 Imperfect efforts to achieve deterministic C++

To help determinism, the Lepton binary is statically linked, and all heap allocations are zeroed before use. However, this setup was insufficient to detect a buffer overrun from incorrect index math in the Lepton model (§ 6.1). We had an additional fail-safe mechanism to detect nondeterminism. Before deploying any version of Lepton, we run it on over a billion randomly selected images (4 billion for the first version), and then decompress each with the same binary and additionally with a single-threaded version of the same code built with gcc using the address sanitizer and undefined-behavior checker [16]. This system detected the nondeterministic buffer overrun after just a few million images were processed and has caught some further issues since (§ 6.7).

5.3 Impact

As of Feb. 16, 2017, Lepton has encoded 203 PiB of images, reducing the size of those images by 46 PiB.

of the PackJPG software, which has various unpublished improvements over the 2007 version.

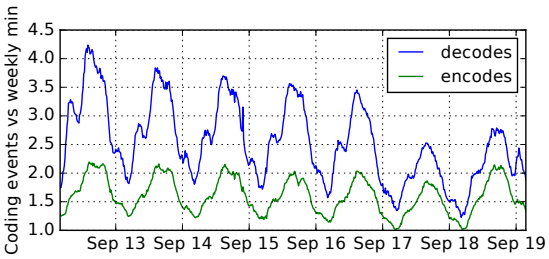


Figure 5: Weekday upload rates are similar to weekends, but weekday download rates of Lepton images are higher.

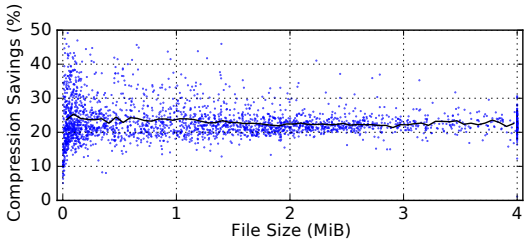


Figure 6: Compression savings on production workloads are uniform across file sizes.

The traffic has been divided between live encode traffic and a steady rate of background encoding of older images. Dropbox has decoded and served 427 billion Lepton-compressed images.

5.4 Workload

Lepton has been in production since April 14[4], and has been on the serving path for all uploads and hundreds of billions of downloads. A typical week can be observed in Figure 5. On the weekends, users tend to produce the same number of photos but sync fewer to their clients, so the ratio of decodes to encodes approaches 1.0. On weekdays, users tend to consume significantly more photos than they produce and the ratio approaches 1.5. Over the last 6 months Dropbox has encoded images with Lepton at between 2 and 12 GiB per second.

When the Lepton encoder accepts a three-color, valid, baseline JPEG, that file compresses down to, on average, 77.31% of its original size (22.69% savings). The savings are uniform across file sizes, as illustrated in Figure 6.

Small images are able to compress well because they are configured with fewer threads than larger images and hence have a higher proportion of the image upon which to train each probability bin. The number of threads per image was selected empirically based on when the overhead of thread startup outweighed the gains of multithreading. The multithreading cutoffs can be noticed in the production performance scatter plot in Figure 7. Because Lepton is streaming, the working set is roughly fixed in size. Profiling the decoder using hardware counters confirmed that the L1 cache is rarely missed. Nearly all L1 delays are due to pipeline stalls, not cache misses.

[4] All dates are in 2016 and times are in UTC.

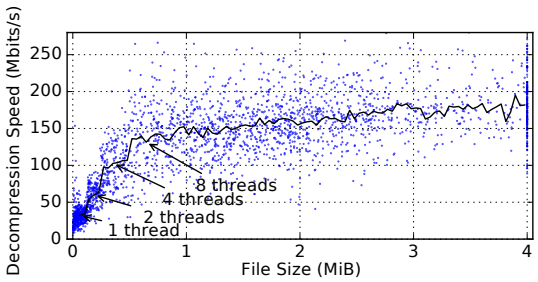


Figure 7: Decompression speed given an input file size.

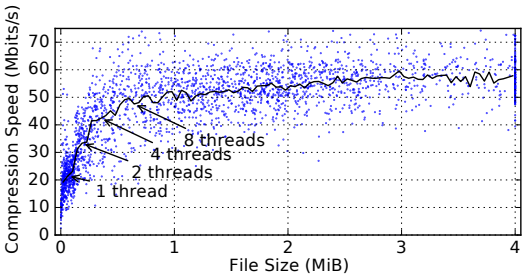


Figure 8: Compression speed given an input file size.

The compression speed, shown in Figure 8, is similar, but it is almost unaffected by the benefit of moving to 8 threads from 4. This is because at 4 threads the bottleneck shifts to the JPEG Huffman decoder, away from the arithmetic coding. This is solved in the Lepton decoder with the Huffman handover words, but the Lepton encoder must decode the original JPEG serially.

5.5 Outsourcing

Blockservers are machines that, among other tasks, respond to requests to store or retrieve data chunks, whether Lepton-compressed JPEGs or Deflate-compressed files. Load balancers, which do not inspect the type of request, randomly distribute requests across the blockserver fleet.

Each blockserver has 16 cores, which means that 2 simultaneous Lepton decodes (or encodes) can completely utilize a machine. However, blockservers are configured to handle many more than 2 simultaneous requests, because non-Lepton requests are far less resource-intensive. Therefore, a blockserver can become oversubscribed with work, negatively affecting Lepton's performance, if it is randomly assigned 3 or more Lepton conversions at once. Without outsourcing, there are an average of 5 encodes/s during the Thursday peak. Individual blockservers will routinely get 15 encodes at once during peak, to the point where there is never a full minute where there isn't at least one machine doing 11 parallel encodes during an hour of peak traffic, as illustrated in the Control line in Figure 9.

We mitigated this problem by allowing overloaded blockservers to "outsource" compression operations to other machines. Inspired by the power of two random choices [14], Lepton will outsource any compression operations that occur on machines that have more than three conversions happening at a time.

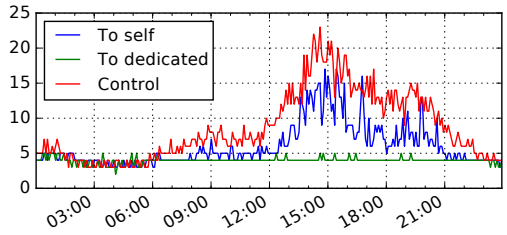


Figure 9: 99th-percentile of concurrent Lepton processes on Sept. 15 with listed outsourcing strategy and threshold=4.

Under normal operation, when the system is under low load, Lepton operates by listening on a Unix-domain socket for files. A file is read from the socket, and the (de)compressed output is written back to the socket. The file is complete once the socket is shut down for writing. When outsourcing, instead of a Unix-domain-socket connection to a local Lepton process, the blockserver instead will make a TCP connection to a machine tagged for outsourcing within the same building in the same datacenter.[5] The overhead from switching from a Unix-domain socket to a remote TCP socket was 7.9% on average.

We have two alternative strategies for selecting machines for outsourcing. The simpler idea was to dedicate a cluster of machines ready to serve Lepton traffic for overloaded blockservers. This cluster is easy to provision to meet traffic demands and can be packed full of work since there are no contending processes on the machines.

Our other strategy was to mark each blockserver as an outsourcing target for other blockservers (denoted "To Self"). The intuition is that in the unlikely case of a current blockserver being overloaded, the randomly chosen outsource destination is likely to be less overloaded than the current machine at the exact contended instant.

5.5.1 Outsourcing Results

Figure 10 illustrates that outsourcing reduces the p99 by 50% at peak from 1.63 s to 1.08 s and the p95 by 25%.

The dedicated cluster reduces the p99 more than simply outsourcing directly to other, busy, blockservers, especially at peak. However, rebalancing traffic within the same cluster of blockservers has the added effect of reducing the p50 as well, since there are fewer hotspots because of the additional load balancing.

5.6 Backfill

Lepton has been configured to use spare compute capacity to gradually compress older JPEG files in storage, a process we call "backfilling." To this end, we developed a small system called DropSpot. DropSpot monitors the

[5] Initially it seemed to make sense logistically to select an outsourcing destination simply in the same metro location as the busy blockserver. However, in measuring the pairwise conversion times, our datacenters in an East Coast U.S. location had a 50% latency increase for conversions happening in a different building or room within, and in a West Coast location, the difference could be as high as a factor of 2.

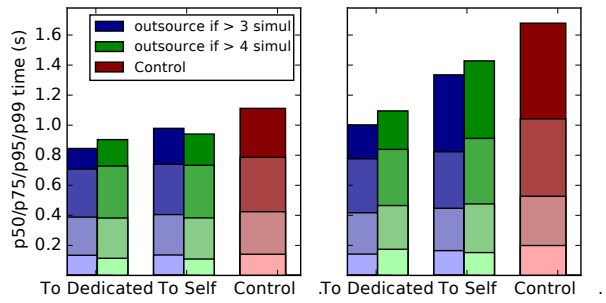


Figure 10: Percentile timings of JPEG compression near to peak traffic (left) and at peak traffic (right) with 2 outsourcing strategies when concurrent Lepton processes exceed a threshold (denoted with bar color) on the local machine.

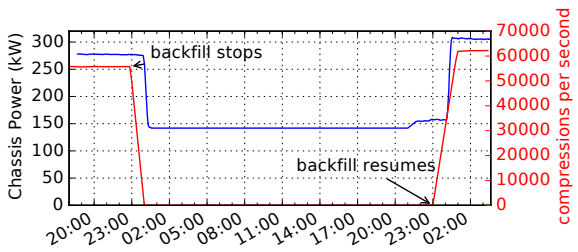


Figure 11: Overall Dropbox datacenter power usage graph on Sept. 26, 2016. During an outage, Lepton backfill was disabled. When shut off, the power usage dropped by 121 kW.

spare capacity in each server room, and when the free machines in a room exceed a threshold, a machine is allocated for Lepton encoding. When too few machines are free, DropSpot releases some.

Wiping and reimaging the machine with the necessary software takes 2-4 hours, so a sufficiently diverse reserve of machines must be available for on-demand use. The backfill system can be run on Amazon spot instances, but our goal of 6,000 encodes per second has been attainable using spare capacity.

In July, all user accounts were added to a sharded table in a database service backed by MySQL. For a Lepton backfill worker to find images to encode, it sends a request to the metadata servers (metaservers) to request work from a randomly chosen shard. The metaserver selects the next 128 user-ids from the table in the corresponding shard. The metaserver scans the filesystem for each user, for all files with names containing the case-insensitive string ".jp" (likely jpeg or jpg). The metaserver builds a list of SHA-256 sums of each 4-MiB chunk of each matching file until it obtains up to 16,384 chunks. The metaserver returns a response with all the SHA-256 sums, the list of user ids to process, and a final, partial user with a token to resume that user. The worker then downloads each chunk and compresses it. It double-checks the result with the gcc address-sanitizer version of Lepton in both single and multithreaded mode, and uploads the compressed version back to Dropbox.

5.6.1 Cost Effectiveness

The cluster has a power footprint of 278 kW and it encodes 5,583 chunks per second (Figure 11). This means that one kWh can be traded for an average of 72,300 Lepton conversions of images sized at an average of 1.5 MB each. Thus, a kWh can save 24 GiB of storage, permanently. The power usage includes three extraneous decodes, which could be tallied as future reads, since the file is redundantly checked three times during backfill.

Imagining a depowered 5TB hard drive costing $120 with no redundancy or checks, the savings from the extra space would be worthwhile as long as a kWh costed less than $0.58. Even in Denmark, where electricity costs about $0.40 per kWh, Lepton would be a worthwhile investment for somewhat balanced read/write load of 3:1. Most countries offer prices between $0.07 and $0.12 per kWh. With cross-zone replication, erasure coding, and regular self-checks, 24 GiB of storage costs significantly more in practice. For instance, buying a year storage for 24 GiB on Amazon S3's Infrequent Access Storage tier, as of February 2017, would cost $3.60 each year, excluding any data read fees, making the actual savings even more clear.

To get the full 5,583 conversions per second, 964 machines are required. This means that each Intel Xeon E5 2650 v2 at 2.6 GHz can backfill 5.75 images per second. This means each server can process 181,500,000 images per year, saving 58.8 TiB of storage. At Amazon S3 Infrequent Access pricing, this would cost $9,031 per year, justifying the savings. Additionally, the storage savings will recur, year over year, while the capital expense of the Xeon for a year will be much less than $9,000 and will depreciate only once.

5.7 Safety Mechanisms

During the initial roll-out, after Lepton was activated for several days after April 14th, all of the several hundred TiB of compressed images had been downloaded in compressed form and decompressed twice in a row, once with a gcc, asan-enabled, Lepton and another time with the default productionized icc Lepton in multithreaded mode.

There are also alerts in place that page a member of the Lepton team if a particular chunk is unable to be decompressed. The construction of this alert required some care(§ 6.6). There is a "playbook entry" for the person on call to immediately disable Lepton.

The shutoff switch operates by placing a file with a predetermined name in /dev/shm, and the Lepton system checks for that file before compressing new chunks. Most Dropbox configuration files take between 15 and 45 minutes to fully deploy, but this mechanism allows a script to populate the file across all hosts that encode Lepton within 30 seconds.

The blockservers also never admit chunks to the stor-

age system that fail to round-trip—meaning, to decode identically to their input. Additionally, all memory pages holding compressed data in memory are protected at the OS level before the round-trip test, and an md5sum is done of the initial compressed file to be compared with the result stored on disk, so the memory contents cannot change due to a user-level bug. Corruptions in compressing will be detected immediately, and the files will be compressed using Deflate instead.

For a file that fails to round-trip, it can be difficult to distinguish between a badly formed/unsupported JPEG file versus a real program bug, but as long as the compressor and decompressor are deterministic, a small level of such failures is acceptable.

For added safety, there is an automated verification process that searches for images that succeeded in a round-trip once but then fail a subsequent round-trip test, or fail when decompressed with the address-sanitizing gcc build of Lepton. If either of those occur, a member of the Lepton team is paged and the failing data is saved. This process has currently gone through over four billion files and has caused four alerts (§ 6.7).

During roll-out of a new Lepton version it will be "qualified" using the automated process over a billion images. Additionally it must be able to decompress another billion images already compressed in the store. Currently a candidate which fails to do so also causes the Lepton team to be paged. This alert has never triggered.

For the first two weeks of ramp-up, the system was completely reversible. Every chunk uploaded with Lepton was concurrently also uploaded to a separate S3 bucket (the "safety net") with the standard Deflate codepath. This means that in the worst case, requests could fail-over directly to the safety net until the affected files were repaired.

Before enabling Lepton, the team did a mock disaster recovery training (DRT) session where a file in a test account was intentionally corrupted and recovered from the safety net. However, we never needed to use this mechanism to recover any real user files.

We have since deleted the safety net and depend on other controls to keep Lepton safe. Our rationale for this was that uploading to a separate bucket causes a performance degradation since all images would upload in the max of latency between Dropbox datacenters and S3, plus associated transaction and storage fees. We may re-enable the safety net during future format upgrades.

Even with the safety net disabled, we believe there are adequate recovery plans in place in case of an unexpected error. Every file that has been admitted to the system with Lepton compression has also round-tripped at least once in order to be admitted. That means that a permanent corruption would expose a hypothetical nondeterminism in the system. But it also means that if the same load/perf circumstances were recreated, the chunk would probably be decodable again with some probability, as it was decoded exactly correctly during the original round-trip check. Thus, with sufficient retries, we would expect to be able to recover the data. That said, it would be a significant problem if there were a nondeterminism in the Lepton system. After 4 billion successful determinism tests, however, we believe the risk is as small as possible.

6 Anomalies at Scale

With a year of Lepton operational experience, there have been a number of anomalies encountered and lessons learned. We share these in the hopes that they will be helpful to the academic community in giving context about challenges encountered in the practical deployment of format-specific compression tools in an exabyte-scale network filesystem.

6.1 Reversed indices, bounds checks and compilers

During the very first qualification of 1 billion files, a handful of images passed the multithreaded icc-compiled check, but those images would occasionally fail the gcc roundtrip check with a segfault. The stack trace revealed the multidimensional statistic-bin index computation was reversed. If deployed, this would have required major backwards compatibility contortions to mimic the undefined C++ behavior as compiled with icc; bins would need to be aliased for certain versions of Lepton.

In response to this discovery, the statistic bin was abstracted with a class that enforced bounds checks on accesses. Consequently, the duration of encodes and decodes are 10% higher than they could be, but bounds checks help guard against undefined behavior.

6.2 Error codes at scale

This table shows a variety of exit codes that we have observed during the first 2 months of backfill.

Success	94.069%	"Impossible"	0.006%
Progressive	3.043%	Abort signal	0.006%
Unsupported JPEG	1.535%	Timeout	0.004%
Not an image	0.801%	Chroma subsample big	0.003%
4 color CMYK	0.478%	AC values out of range	0.001%
>24 MiB mem decode	0.024%	Roundtrip failed	0.001%
>178 MiB mem encode	0.019%	OOM kill	10^{-5}%
Server shutdown	0.010%	Operator interrupt	10^{-6}%

The top 99.9864% of situations were anticipated: from graceful shutdown, to deciding not to encode JPEG files that consist entirely of a header, to unsupported Progressive and CMYK JPEGs, and chroma subsampling that was larger than the slice of framebuffer in memory.

The Lepton program binary could process these types of images, e.g., by allocating more memory, an extra model for the 4th color channel, or sufficient memory on decode to keep the progressive image resident. However, for simplicity, these features were intentionally disabled in Dropbox as they account for a small fraction of files.

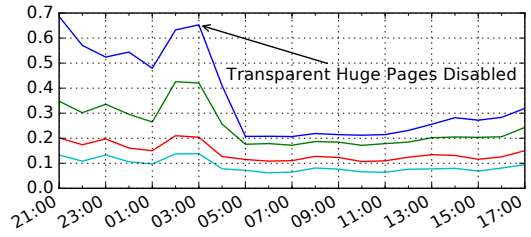


Figure 12: Hourly p99/p95/p75/p50 latency for decodes. Transparent huge pages disabled April 13 at 03:00.

Some codes were unexpected, e.g., incorrect thread protocol communication (listed as "Impossible"), or "Abort signal", since `SECCOMP` disallows `SIGABRT`. By contrast, a small level of "Roundtrip failed" was expected, largely because of corruptions in the JPEG file (e.g., runs of zeroes written by hardware failing to sync a file) that cannot always be represented in the Lepton file format.

6.3 Poor p95 latency from Huge Pages

During the Lepton roll-out, after the qualification round, a significant fraction of machines had significantly higher average and p99 latency, and could take 2–3× as long as the isolated benchmarks. It was even possible to have a decode take 30 seconds to even begin processing. The time would elapse before a single byte of input data was read. Reboots could sometimes alleviate the issue, but on the affected machines it would come back. Also, when services on the machine were halted and benchmarks run in isolation, the problem disappeared altogether and the machine performed as expected.

On affected machines, performance counters attributed 15–20% of the time to the kernel's page-table routines.

9.79%	[*kernel*]	[*k*]	isolate_migratepages_range
4.04%	[*kernel*]	[*k*]	copy_page_range
3.16%	[*kernel*]	[*k*]	set_pfnblock_flags_mask

These kernel routines implicated transparent huge pages (THP), and each affected machine had THP enabled [*Always*], but most unaffected machines had them disabled. Disabling THP solved the issue (Figure 12).

Additionally, when THP is enabled, Linux continuously defragments pages in an attempt to build a full 2 MiB page of free memory for an application requesting large ranges of data. Since Lepton requests 200 MiB of space at initialization time, with no intent to use more than 24 MiB for decodes, Linux may prepare a significant number of huge pages for use, causing the process to be blocked during defragmentation. These pages are consumed without penalty over the next 10 decodes, meaning that the p95 and p99 times are disproportionately affected by the stall (compared with the median times).

6.4 Boiling the frog

Currently, the Lepton system decodes about 1.5× to twice as many files as it encodes. However, during the initial roll-out, months before the backfill system, the ratio of

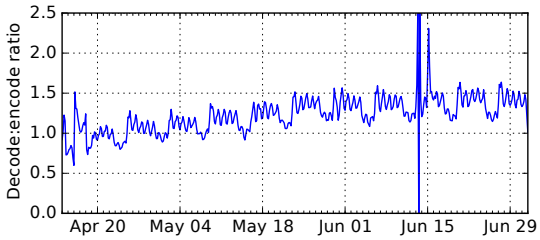


Figure 13: Lepton decode:encode ratio on the serving path.

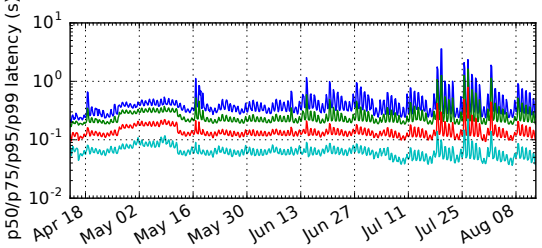


Figure 14: Decode timing percentiles, starting with the roll-out and building up over months to multi-second p99s.

decodes to encodes was much less than 1.0, since each old photo was compressed using Deflate, not Lepton, and only new photos need a Lepton decompress. This can be seen in the historical graph of the decode:encode ratio over time in Figure 13. Akin to "boiling the frog," it was not obvious that the actual hardware requirements would be significantly higher than those needed from having reached 100% of users in the first weeks.

To react to these new requirements for decodes, we built the outsourcing system (§ 5.5). But until that system rolled out, for several months, at peak, our 99th-percentile decode time was in the seconds, as seen in Figure 14.

6.5 Dropbox camera upload degradation

Before removal of the safety net, each image would be uploaded compressed to the Dropbox store and uncompressed to the S3 safety net. During maintenance in one East-Coast datacenter, each top of rack switch required a reboot. Traffic was rerouted to another datacenter. The transition was going well, but on June 13 at 8:40:25, once most traffic had moved to the new location, S3 "put" operations began to fail sporadically from truncated uploads. The safety-net feature was writing more data to S3 from the new location than *all of the rest of Dropbox combined*, and the capacity of our S3 proxy machines was overtaxed by the safety-net mechanism.

For uploads, the availability dropped to 94% for the 9 minutes required to diagnose the situation, and camera uploads from phones were disproportionately affected, as mobile devices are a very common means for users to capture images. Availability of this service dropped to 82%, since each photograph upload required a write to the safety net. Once the situation was identified, Lepton encodes were disabled in 29 seconds, using the shutoff

switch (§ 5.7) in /dev/shm, stopping extra traffic to S3. Traffic returned to normal at 8:49:54.

An irony emerged: a system we designed as a belt-and-suspenders safety net ended up causing our users trouble, but has never helped to resolve an actual problem.

6.6 Decodes that exceed the timeout window

With thousands of servers decoding chunks, there are often unhealthy systems that are swapping, overheating, or broken. These can become stuck during a Lepton decode and time out. Because these events happen regularly, they must be investigated automatically without involving a human operator.

Instead, any decode exceeding a timeout is uploaded to an S3 queue bucket. Chunks in this queue bucket are decompressed on an isolated, healthy cluster without a timeout using the gcc-asan as well as the icc build of Lepton. If the chunk is successfully decoded 3 times in a row with each build, then the chunk is deleted from the bucket. If any of those decodes fails, a human is signaled.

6.7 Alarm pages

As of this submission, anomalies in the Lepton system have caused an on-call engineer to be paged four times.

Assert failed in sanitizing build only. The first alarm occurred just days after Lepton was activated for 0.1% of users, on April 8. When reconstructing the Huffman coded data, each thread asserts that the number of bytes produced matches the number of bytes decoded on the initial compression. A file tripped this assert in the gcc-asan build that was disabled for the icc production build, so the icc build admitted the file.

The solution was to compile Lepton with all meaningful asserts enabled and to check whether any of the existing 150 TiB of images tripped the assert. Luckily no other files triggered the assert. We deployed stricter code that will not admit such files.

Single- and multi-threaded code handled corrupt JPEGs differently. The second alarm was triggered on May 11 because of a bug in the single-threaded code. The single-threaded decoder wrote all output directly to the file descriptor, whereas in the multithreaded decoder, each thread wrote to a fixed sized memory area. When the JPEG was sufficiently corrupt, the size would be incorrectly computed, but the writes to the memory area would be truncated in multi-threaded mode, yet the direct writes to the stream would be unbounded in single-thread mode. The fix was to make sure single-threaded decodes bounded their writes to the stream as if it were a fixed memory region.

After open-source release, fuzzing found bugs in parser handling of corrupt input. The third alarm was caused by a security researcher [7], who fuzzed the open-source release of Lepton and found bugs in the uncmpjpg JPEG-parsing library that Lepton uses. The library did not validate that the Huffman table had sufficient space for the data. Uncmpjpg would overwrite global memory past the array with data from the untrusted input. A similar bug existed in uncmpjpg's quantization table index, which also opened up a buffer overrun. The response was to replace every raw array with a bounds-checked `std::array`, to avoid similar attacks in the future. It was unfortunate that we did not apply this philosophy after the earlier "reversed indices" incident (§ 6.1). Fortunately, the deployed system is protected with `SECCOMP`, preventing escalation of privileges.

Accidental deployment of incompatible old version. The final alarm was the result of a series of operational mistakes. On Dec. 12, 2016, a new team member was deploying Lepton on some blockservers. The internal deployment tool asks the operator to specify the hash of a Lepton build to deploy. These builds have all been "qualified," meaning they successfully compressed and then decompressed a billion JPEGs with both optimized and sanitizing decoders, yielding identical results to the input.

Our historical practice has been to retain earlier "qualified" builds as eligible for deployment, so that Lepton can be rolled back if necessary. However, because Lepton's file format has evolved over time, the earliest qualified builds are not compatible with more recent versions. When features were added, an older decoder may not be able to decode a newer file. When Lepton's format was made stricter, an older encoder may produce files that are rejected by a newer decoder. At the time of such upgrades, we searched for and re-encoded JPEG files in Dropbox as necessary, but we did not remove the older software from the list of qualified builds.

Typically, our team members deployed Lepton to blockservers by specifying the hash of the most recent qualified build in the deployment tool. However, our documentation did not properly inform the new employee of this practice, and they simply left the field blank. This caused the deployment tool to use an internal default value of the hash, which had been set when Lepton was first deployed and never updated. As a result, the very first qualified version of Lepton was accidentally deployed on some blockservers.

The first warning sign was availability dropping to 99.7% for upload and download endpoints. This was due to the oldest qualified Lepton code being unable to decode some newly compressed images because of minor additions to the format. An additional alarm was triggered after other blockservers (ones that did not receive the bad configuration change) found themselves unable to decode some files that had been written by blockservers that did receive the change.

As operators were occupied trying to roll back the

configuration change, it took two hours before Lepton was disabled, during which time billions of files were uploaded. We performed a scan over all these files, decoding and then re-encoding them if necessary into the current version of the Lepton file format. Ultimately, 18 files had to be re-encoded.

This was an example of a number of procedures gone wrong. It confirms the adage: we have met the enemy and he is us. The incident has caused us to reconsider whether a "qualified" version of Lepton ought to remain eternally qualified for deployment, the behavior and user interface of the deployment tools, and our documentation and onboarding procedures for new team members.

7 Limitations and Future Work

Lepton is currently deployed on Dropbox's back-end file servers, and is transparent to client software. In the future, we intend to move the compression and decompression to client software, which will save 23% in network bandwidth when uploading or downloading JPEG images.

Lepton is limited to JPEG-format files, which account for roughly 35% of the Dropbox filesystem. Roughly another 40% is occupied by H.264 video files, many of which are encoded by fixed-function or power-limited mobile hardware that does not use the most space-efficient lossless compression methods. We intend to explore the use of Lepton-like recompression for mobile video files.

8 Conclusion

Lepton is an open-source system that compresses JPEG images by 23% on average. It has been deployed on the production Dropbox network filesystem for a year and has so far compressed more than 150 billion user JPEG files that accounted for more than 203 PiB. Lepton was designed to be deployable on a distributed file-serving backend where substrings of a file must be decodable independently, with low time-to-first-byte and time-to-last-byte. The system demonstrates new tradeoffs between speed, compression efficiency, and deployability in the context of a large-scale distributed filesystem back-end.

In a year of production use and hundreds of billions of downloads, deployment has been relatively smooth. We have never been unable to decode a stored file. The issues we have encountered have involved human error and procedural failures, non-obvious ways in which the system created load hotspots, and difficulties in ensuring deterministic behavior from a highly optimized C++ program processing untrusted input from diverse sources. We have shared a number of deployment case studies and anomalies in the hopes that they will be helpful to the academic community in giving context about challenges encountered in the practical deployment of format-specific compression tools at a large scale.

9 Acknowledgments

We would like to thank the reviewers and NSDI program committee for their detailed comments and suggestions.

This work was funded by Dropbox. DH, KE, and CL are Dropbox employees. KW's participation was as a paid consultant and was not part of his Stanford duties or responsibilities.

We would like to thank Jongmin Baek, Sujay Jayakar, and Mario Brito for their ideas on the Lepton approach, and Preslav Le, David Mah, James Cowling, Nipunn Koorapati, Lars Zornes, Rajat Goel, Bashar Al-Rawi, Tim Douglas, and Oleg Guba for their assistance and guidance on deploying and rolling out Lepton to Dropbox infrastructure.

This project would not have been possible without the continuous support and encouragement from Dropbox leadership including Dimitry Kotlyarov, Nahi Ojeil, Bean Anderson, Ziga Mahkovec, David Mann, Jeff Arnold, Jessica McKellar, Akhil Gupta, Aditya Agarwal, Arash Ferdowsi, and Drew Houston.

References

[1] Lepton source repository. http://github.com/dropbox/lepton.

[2] AAS, J. Introducing the 'mozjpeg' project, 2014. https://blog.mozilla.org/research/2014/03/05/introducing-the-mozjpeg-project/.

[3] ALAKUIJALA, J., AND SZABADKA, Z. Brotli compressed data format. RFC 7932, RFC Editor, July 2016. https://tools.ietf.org/html/rfc7932.

[4] BANKOSKI, J., KOLESZAR, J., QUILLIO, L., SALONEN, J., WILKINS, P., AND XU, Y. VP8 Data Format and Decoding Guide section 13.2. RFC 6386, RFC Editor, November 2011. https://tools.ietf.org/html/rfc6386#section-13.2.

[5] COLLET, Y., AND TURNER, C. Smaller and faster data compression with Zstandard, 2016. https://code.facebook.com/posts/1658392934479273/smaller-and-faster-data-compression-with-zstandard/.

[6] DEUTSCH, P. DEFLATE compressed data format specification version 1.3. RFC 1951, RFC Editor, May 1996. https://tools.ietf.org/html/rfc1951.

[7] GRASSI, M. Some memory corruptions in lepton. https://github.com/dropbox/lepton/issues/26.

[8] HUFFMAN, D. A. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE 40*, 9 (1952), 1098–1101.

[9] Independent JPEG Group. http://www.ijg.org/.

[10] *Information technology — Digital compression and coding of continuous-tone still images: Requirements and guidelines*, September 1992. ITU-T Rec.

T.81 and ISO/IEC 10918-1:1994 (https://www.w3.org/Graphics/JPEG/itu-t81.pdf).

[11] LAKHANI, G. DCT coefficient prediction for JPEG image coding. In *2007 IEEE International Conference on Image Processing* (San Antonio, TX, USA, Sept 2007), vol. 4, pp. IV–189 to IV–192.

[12] MAHONEY, M. Paq: Data compression programs, 2009. http://mattmahoney.net/dc/.

[13] MERRITT, L. JPEGrescan: losslessly shrink any JPEG file, 2013. https://github.com/kud/jpegrescan.

[14] MITZENMACHER, M., RICHA, A. W., AND SITARAMAN, R. The power of two random choices: A survey of techniques and results. In *Handbook of Randomized Computing* (2000), Kluwer, pp. 255–312.

[15] PAVLOV, I. LZMA SDK, 2007. http://www.7-zip.org/sdk.html.

[16] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 28–28.

[17] STIRNER, M., AND SEELMANN, G. Improved redundancy reduction for JPEG files. In *Proceedings of the 2007 Picture Coding Symposium* (Lisbon, Portugal, 2007).

[18] TEUHOLA, J. A compression method for clustered bit-vectors. *Information processing letters 7*, 6 (1978), 308–311.

[19] VOORMEDIA B.V. TinyJPG. https://tinyjpg.com/.

[20] WALLACE, G. K. The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics 38*, 1 (1992), 34.

[21] WANG, X., CHEN, H., CHEUNG, A., JIA, Z., ZELDOVICH, N., AND KAASHOEK, M. F. Undefined behavior: what happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems* (Seoul, Republic of Korea, 2012), ACM, p. 9.

[22] XU, X., AICHTA, Z., GOVINDAN, R., LLOYD, W., AND ORTEGA, A. Context adaptive thresholding and entropy coding for very low complexity JPEG transcoding. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (Shanghai, China, 2016), IEEE, pp. 1392–1396.

[23] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*, 3 (1977), 337–343.

A Appendix

A.1 File Format

Magic Number (0xcf, 0x84)	(2 bytes)
Version (0x01)	(1 byte)
Skip serializing header? (Y ‖ ℕ)	(1 byte)
Number of Thread Segments	(4 bytes)
Truncated Git Revision	(12 bytes)
Output File Size	(4 bytes)
Zlib Data Size	(4 bytes)

Zlib Data

JPEG Header Size (4 bytes)
JPEG Header
Pad Bit (0 ‖ 0xFF)
Per-Thread Segment Information

Thread Segment Vertical Range	(2 bytes)
Size of Thread Segment Output	(4 bytes)
Huffman Handover Word	(2 bytes)
DC per channel	(8 bytes)

Number of RST markers
Total Number of 8x8 JPEG Blocks per channel
Arbitrary data to prepend to the output
Arbitrary data to append to the output

Interleaved Arithmetic Coding Section

Thread Segment Id
Length (256‖4096‖65536‖arbitrary)
Arithmetic coded data

Thread Segment Id
Length (256‖4096‖65536‖arbitrary)
Arithmetic coded data

...

May be repeated many times per thread segment

...

A.2 Using prior information to predict and encode DCT coefficients

Each 8x8 JPEG block has 49 2D DCT coefficients, 14 1D DCT coefficients, and one DC coefficient (§ 3.3). Lepton encodes each kind of coefficient using the same Exp-Golomb code (unary exponent, then sign bit, then residual bits) but with different methods for indexing the adaptive arithmetic code's bins. Prior information, such as neighboring blocks, is used to predict bin indices; higher correlation between the predicted indices and the actual coefficient values yields better compression ratios.

A.2.1 Predicting the 7x7 AC coefficients

Lepton first encodes the number of non-zero coefficients in the block, $n \in \{0,\ldots,49\}$, by emitting 6 bits. Since

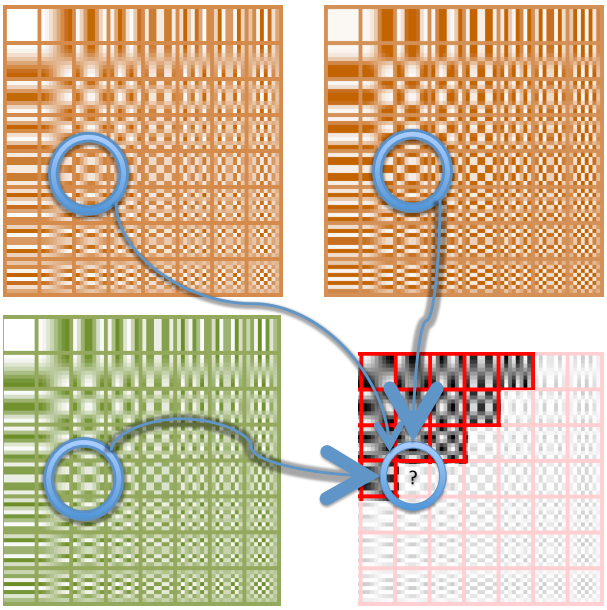


Figure 15: The neighboring coefficients at the same coordinates are averaged to predict the 7x7 coefficient.

the number of non-zero coefficients in the above and left blocks approximately predicts n, the bins are indexed by $\lfloor \log_{1.59}(\frac{n_A+n_L}{2}) \rfloor \in \{0,\ldots,9\}$. The bin for each bit is further indexed by the previously decoded bits, so that the total number of bins (for encoding n) is $10 \times (2^6-1)$.

The 7x7 coefficients are encoded in zigzag order [20], which yields a 0.2% compression improvement over raster-scan order. For each coefficient F, we compute a weighted average of the corresponding coefficient from the above, left, and above-left blocks (Figure 15): $\bar{F} = \frac{1}{32}(13F_A + 13F_L + 6F_{AL})$. Each coefficient is Exp-Golomb encoded using bins indexed by $(\lfloor \bar{F} \rfloor, \lfloor \log_{1.59} n \rfloor)$. Each time a non-zero coefficient is encoded, n is decremented; the block is finished when $n = 0$.

A.2.2 Predicting the 7x1 and 1x7 AC coefficients

The 7x1 and 1x7 coefficients represent image variation purely in the horizontal and vertical directions. Lepton encodes them similarly to the 7x7 coefficients, but instead of predicting each coefficient using a weighted average, we use a more sophisticated formula inspired by Lakhani [11]. When encoding the 1x7 coefficients, Lepton has already encoded the 7x7 coefficients, as well as the full block to the left of the current block. We combine this prior information with the additional assumption that the image pixel values are continuous across the block edge — i.e., that $P_L(7,y) \approx P(0,y)$.

The block's pixels are defined to be a linear combination of orthonormal DCT basis functions B:

$$P(x,y) = \sum_{u=0}^{7}\sum_{v=0}^{7} B(x,u)B(y,v)F_{uv}$$

Written as matrices, $P = B^T F B$ where $B^T B = 1$. The

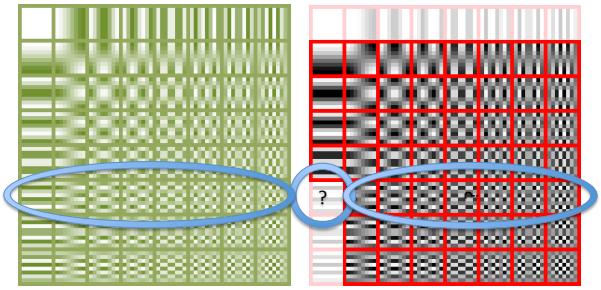


Figure 16: An entire row (or column) of coefficients may be used to predict an edge DCT coefficient.

continuity assumption can then be rewritten as:

$$\begin{aligned} e_7 B^T L B &\approx e_0 B^T F B \\ e_7 B^T L &\approx e_0 B^T F \end{aligned}$$

The left side is fully known from the block to the left, while the right side is a linear combination of the known 7x7 coefficients and the unknown 1x7 coefficients, as shown in Figure 16.

$$\sum_{u=0}^{7} B_{7u} L_{uv} \approx B_{00} F_{0v} + \sum_{u=1}^{7} B_{0u} F_{uv}$$

We solve for the unknowns to predict F_{0v}:

$$\bar{F}_{0v} = \frac{1}{B_{00}} \left(\sum_{u=0}^{7} B_{7u} L_{uv} - \sum_{u=1}^{7} B_{0u} F_{uv} \right)$$

The predicted $\bar{F}_{0v}$ is quantized to 7 bits and concatenated with the non-zero count as the bin index for encoding the true coefficient F_{0v}.

A.2.3 Predicting the DC coefficient

With all 63 of the AC coefficients known, the last block element to be encoded is the DC coefficient. Instead of encoding this value directly, Lepton instead predicts a DC coefficient and encodes the delta (the DC error term) between the true value and the prediction. To make this prediction, Lepton first computes the 8x8 pixel block (up to the constant DC shift) from the known AC coefficients using inverse DCT.

A first-cut prediction, illustrated in Figure 17 (left), might be to compute the DC value that minimized the differences between all 16 pairs of pixels at the borders between the current 8x8 block and each of its left and above neighbors. If we average the median 8 pairs and discard the 8 outlier pairs, this technique compresses the DC values by roughly 30% versus baseline JPEG.

We can improve upon this prediction by observing that images tend to have smooth gradients; for example, the sky fades from blue to orange towards the horizon during a sunset. Lepton interpolates the pixel gradients smoothly between the last two rows of the previous block and the

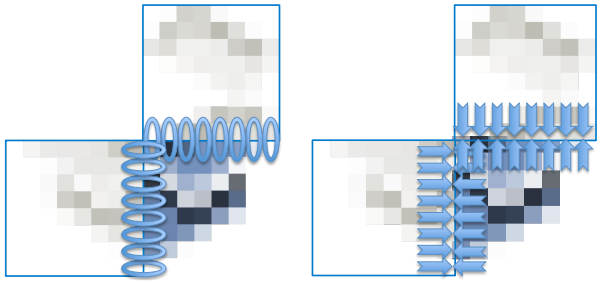


Figure 17: Left: illustrates minimizing differences between pairs of pixels by predicting the DC value for the block being decoded (shaded in blue). DC adds a constant shift to all colors in the blue 8x8 block. Right: illustrates using gradients to interpolate colors between the blue block and its neighbors.

first two rows of the current block, as illustrated in Figure 17 (right). For each border pair, we predict the DC value that would cause the two gradients to meet seamlessly. We finally encode the DC error term between the true DC coefficient and the average of all 16 predictions. Lepton estimates the DC prediction confidence by subtracting the maximum and minimum predictions (out of 16), and uses this value as the bin index when Exp-Golomb encoding the error term. The combination of these techniques yields another 10% compression improvement over the first cut: the final compression ratio is 40.1% $\pm$ 8.7 better than the JPEG baseline.

A.3 Common JPEG Corruptions

In the Lepton qualification process, there were several common JPEG anomalies that would trigger roundtrip failures.

Most prevalently, JPEG files sometimes contain or end with runs of zero bytes. Likely these are caused by failures for an image editing tool or hard disk to sync pages to disk before a user depowered their machine. Many such images will successfully roundtrip with Lepton since zero describes valid DCT data.

However, RST markers foil this fortuitous behavior, since RST must be generated at regular block intervals in image space. Ironically the very markers that were designed to recover from partial corruption instead caused it for Lepton. For affected files, during these zero runs, the RST markers, beginning with a signature 0xff, would not be present. However, Lepton blindly uses the RST frequencies in the header to insert them at regular intervals irrespective of the bytes in the original file. The fix for these zero-filled files regions at the end was to add a RST count to the Lepton header, so that Lepton could cease automatically inserting RST markers after the last one was recorded in the original file.

The RST solution did not fix issues with zero runs appearing in the middle of a scan, since one count cannot describe both the start and stop of RST insertions. These corruptions manifest themselves as roundtrip failures.

A very common corruption was arbitrary data at the end of the file. There are documented cases of cameras producing a TV-ready interlaced image file in a completely different format at the end of JPEG files produced. This let old cameras display images directly to TVs. One of the authors had a series of files where two JPEGs were concatenated, the first being a thumbnail of the second. For these, the compression ratio is less, since Lepton only applies to the thumbnail, but they do roundtrip.

Likewise, the JPEG specification does not mention whether partial bytes, filled with fewer than 8 bits, must be padded with zeros or with ones. Most encoders pick a pad bit and use it throughout. Lepton stores this bit in the header.

Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage

Mihir Nanavati Jake Wires Andrew Warfield
Coho Data and University of British Columbia

Abstract

The performance characteristics of modern non-volatile storage devices have led to storage becoming a shared, rack-scale resource. System designers have an imperative to rethink existing storage abstractions in light of this development. This paper proposes a lightweight storage abstraction that encapsulates full-system hardware resources and focuses only on isolating and sharing storage devices across multiple, remote tenants. In support of this, it prototypes a shared-nothing runtime capable of managing hardware and scheduling this abstraction in a manner that avoids performance interference and preserves the performance of the storage devices for tenants.

1 Introduction

Rack-scale storage architectures such as Facebook's Lightning [53] and EMC's DSSD [17] are dense enclosures containing storage class memories (SCMs)[1] that occupy only a few units of rack space and are capable of serving millions of requests per second across petabytes of persistent data. These architectures introduce a tension between efficiency and performance: the bursty access patterns of applications necessitate that storage devices be shared across multiple tenants in order to achieve efficient utilization [39], but the microsecond-granularity access latencies of SCMs renders them highly sensitive to software overheads along the datapath [12, 64].

This tension has forced a reconsideration of storage abstractions and raised questions about where functionality, such as virtualization, isolation, and redundancy, should be provided. How should these abstractions evolve to support datacenter tenants without compromising efficiency, performance, or overall system complexity?

Decibel is a thin virtualization platform, analogous to a processor hypervisor, designed for rack-scale storage, that demonstrates the ability to provide tenants with flexible, low-latency access to SCMs. Its design is motivated by the following three observations:

1. Device-side request processing simplifies storage implementations. By centralizing the control logic necessary for multi-tenancy at processors adjacent to storage devices, traditional storage systems impose latency overheads of several hundreds of microseconds to few milliseconds under load [39]. In an attempt to preserve device performance, there has been a strong trend towards bypassing the CPU altogether and using hardware-level device passthrough and proprietary interconnects to present SCMs as *serverless storage* [12, 27, 3].

Serverless storage systems throw the proverbial baby out with the bathwater – eliminating the device-side CPU from the datapath also eliminates an important mediation point for client accesses and shifts the burden of providing datapath functionality to client-based implementations [4, 11] or to the devices themselves [71, 27, 1]. For isolation, in particular, client implementations result in complicated distributed logic for co-ordinating accesses [27] and thorny questions about trust.

2. Recent datacenter infrastructure applications have encompassed functionality present in existing feature-rich storage abstractions. In addition to virtualizing storage, storage volumes provide a rich set of functionality such as data striping, replication, and failure resilience [51, 18, 9, 28]. Today, scalable, cloud-focused *data stores* that provide persistent interfaces, such as key-value stores, databases, and pub/sub systems, increasingly provide this functionality as part of the application; consequently, the provision of these features within the storage system represents a duplication of function and risks introducing both waste and unnecessary overheads.

3. Virtualizing only the capacity of devices is insufficient for multi-tenancy. Extracting performance from SCMs is extremely compute-intensive [72, 12, 50] and sensitive to cross-core contention [6]. As a result, storage systems require a system-wide approach to virtualization and must ensure both adequate availability of compute, network, and storage resources for tenant requests, and the ability to service these requests in a contention-free manner. Further, unlike traditional storage volumes that do not adequately insulate tenants from performance interference [39], the system must provide tenants with predictable performance in the face of multi-tenancy.

These observations guide us to a minimal storage abstraction that targets isolation and efficient resource sharing for disaggregated storage hardware. Decibel introduces *Decibel volumes* (referred to as *dVols* for short): vertical slices of the storage host that bind SCMs with the compute and network resources necessary to service

[1] We use the term storage class memory through the rest of the paper to characterize high performance PCIe-based NVMe SSDs and NVDIMM-based persistent storage devices.

tenant requests. As both the presentation of fine-grained storage abstractions, such as files, objects, and key-value pairs, and datapath functionality, such as redundancy and fault-tolerance, have moved up the stack, dVols provide a minimal consumable abstraction for shared storage without sacrificing operational facilities such as transparent data migration.

To ensure microsecond-level access latencies, Decibel prototypes a runtime that actively manages hardware resources and controls request scheduling. The runtime partitions hardware resources across cores and treats dVols as schedulable entities, similar to threads, to be executed where adequate resources are available to service requests. Even on a single core, kernel scheduling policies may cause interference, so Decibel completely bypasses the kernel for both network and storage requests, and co-operatively schedules request processing logic and device I/O on a single thread.

Decibel is evaluated using a commodity Xeon server with four directly-connected enterprise PCIe NVMe drives in a single 1U chassis. Decibel presents storage to remote tenants over Ethernet-based networking using a pair of 40GbE NICs and achieves device-saturated throughputs with a latency of 220-450µs, an overhead of approximately 20-30µs relative to local access times.

2 Decibel and dVols

Scalable data stores designed specifically for the cloud are important infrastructure applications within the datacenter. Table 1 lists some popular data stores, each of which treats VM or container-based nodes as atomic failure domains and handles lower-level network, storage, and application failures uniformly at the node level. As a result, several of these data stores recommend deploying on "ephemeral", locally-attached disks in lieu of reliable, replicated storage volumes [47, 40, 14].

These systems are designed to use simple local disks because duplicating functionality such as data redundancy at the application and storage layers is wasteful in terms of both cost and performance; for example, running a data store with three-way replication on top of three-way replicated storage results in a 9x write amplification for every client write. Further, running a replication protocol at multiple layers bounds the latency of write requests to the latency of the slowest device in the replica set. The desire to minimize this latency has led to the development of persistent key-value stores, such as LevelDB and RocksDB, that provide a simple, block-like storage abstraction and focus entirely on providing high performance access to SCMs.

The dVol is an abstraction designed specifically for rack-scale storage in response to these observations. As a multi-tenant system, Decibel faces challenges similar to a virtual machine monitor in isolating and sharing an extremely fast device across multiple tenants and benefits from a similar lightweight, performance-focused approach to multiplexing hardware. Correspondingly, a dVol is a schedulable software abstraction that encapsulates the multiple hardware resources required to *serve* stored data. In taking an end-to-end view of resource sharing and isolation rather than focusing only on virtualizing storage capacity, dVols resemble virtual machines to a greater degree than traditional storage volumes.

Application	Backend	FT	Ephem	Year
Key-Value Stores				
Riak	LevelDB	Y	Y	2009
Voldemort	BerkeleyDB	Y	N	2009
Hyperdex	File	Y	Y	2011
Databases				
Cassandra	File	Y	Y	2008
MongoDB	RocksDB	Y	N	2009
CockroachDB	RocksDB	Y	Y	2014
Pub/Sub Systems				
Kafka	File	Y	Y	2011
Pulsar	BookKeeper	Y	Y	2016

Table 1: **Examples of cloud data stores.** *FT* denotes that replication and fault-tolerance are handled within the data store and storage failures are treated as node failures. *Ephem* indicates that users are encouraged to install data stores over local, "ephemeral" disks – Voldemort and Mongo suggest using host-level RAID as a convenience for recovering from drive failures. Backend is the underlying storage interface; all of these systems assume a local filesystem such as ext4, but several use a library-based storage abstraction over the file system API.

In borrowing from VMs as a successful abstraction for datacenter computation, dVols provide the following properties for storage resources:

Extensible Hardware-like Interfaces: dVols present tenants with an interface closely resembling a physical device and so avoid restricting application semantics. dVols also offer tenants the ability to offload functionality not directly supported by SCMs. For example, dVols support atomic updates [26, 13] and compare-and-swap [67] operations.

Support for Operational Tasks: Decoupling storage from the underlying hardware provides a valuable point of indirection in support of datacenter resource management. In virtualizing physical storage, dVols support operational tasks such as non-disruptive migration and provide a primitive for dynamic resource schedulers to optimize the placement of dVols.

Visibility of Failure Domains: Unlike traditional vol-

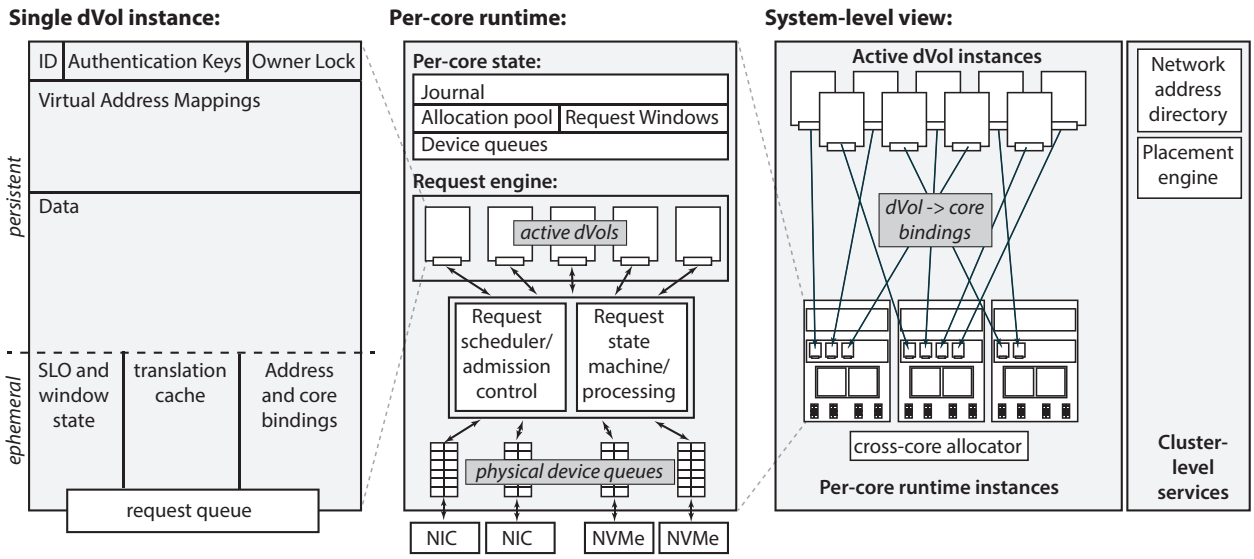


Figure 1: dVol and per-core runtime architecture in Decibel

umes that abstract information about the underlying hardware away, dVols retain and present enough device information to allow applications to reason about failure domains and to appropriately manage placement across hosts and devices.

Elastic Capacity: SCMs are arbitrarily partitioned into independent, non-contiguous dVols at runtime and, subject to device capacity constraints, can grow and shrink during execution without causing device fragmentation and a corresponding wastage of space.

Strong Isolation and Access Control: As multi-tenant storage runs the risk of performance interference due to co-location, dVols allow tenants to specify service-level objectives (SLOs) and provide cross-tenant isolation and the throughput guarantees necessary for data stores. In addition, dVols include access control mechanisms to allow tenants to specify restrictions and safeguard against unauthorized accesses and information disclosure.

As the basis for a scalable cloud storage service, Decibel represents a point in the design space that is between host-local ephemeral disks on one hand, and large-scale block storage services such as Amazon's Elastic Block Store (EBS) on the other. Like EBS volumes, dVols are separate from the virtual machines that access them, may be remapped in the face of failure, and allow a greater degree of utilization of storage resources than direct access to local disks. However, unlike volumes in a distributed block store, each dVol in Decibel is stored entirely on a single physical device, provides no storage-level redundancy, and exposes failures directly to the client.

3 The Decibel Runtime

Decibel virtualizes the hardware into dVols and multiplexes these dVols onto available physical resources to ensure isolation and to meet their performance objectives. Each Decibel instance is a single-host runtime that is responsible solely for abstracting the remote, shared nature of disaggregated storage from tenants. Decibel can provide ephemeral storage directly to tenants or act as a building block for a larger distributed storage system where multiple Decibel instances are combined to form a network filesystem or an object store [65, 15].

Decibel's architecture is shown in Figure 1: it partitions system hardware into independent, shared-nothing per-core runtimes. As achieving efficient resource utilization requires concurrent, wait-free processing of requests and needs to eliminate synchronization and coherence traffic that is detrimental to performance, Decibel opts for full, top-to-bottom system partitioning. Each per-core runtime has exclusive access to a single hardware queue for every NIC and SCM in the system and can access the hardware without requiring co-ordination across cores.

Decibel relies on kernel-bypass libraries to partition the system and processes I/O traffic within the application itself; a network stack on top of the DPDK [30] and a block layer on top of the SPDK [32] provide direct access to network and storage resources. Each per-core runtime operates independently and is uniquely addressable across the network. Execution on each core is through a single kernel thread on which the runtime co-operative schedules network and storage I/O and request processing along with virtualization and other datapath services.

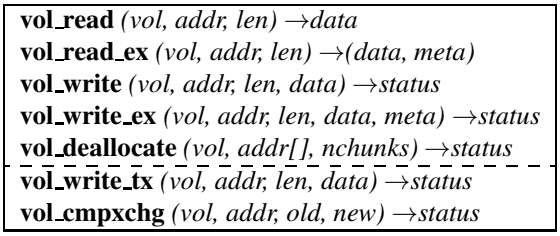


Figure 2: Datapath Interfaces for dVols. The last two provide functionality not directly provided by SCMs in hardware.

Each per-core runtime services requests from multiple tenants for multiple dVols, while each dVol is bound to a single core. The mapping from dVols to the host and core is reflected in the network address directory, which is a separate, global control path network service. As self-contained entities, dVols can be migrated across cores and devices within a host or across hosts in response to changes in load or performance objectives.

By forcing all operations for a dVol to be executed serially on a single core, Decibel avoids the contention overheads that have plagued high-performance concurrent systems [8, 10]. Binding dVols to a single core and SCM restricts the performance of the dVol to that of a single core and device, forcing Decibel to rely on client-side aggregation where higher throughput or greater capacity are required. We anticipate that the runtime can be extended to split existing hot dVols across multiple cores [2] to provide better performance elasticity.

Clients provision and access dVols using a client-side library that maps client interfaces to remote RPCs. The library also handles interaction with the network address directory, allowing applications to remain oblivious to the remote nature of dVols. Legacy applications could be supported through a network block device driver; however, this functionality is currently not provided.

3.1 Virtual Block Devices

Storage virtualization balances the need to preserve the illusion of exclusive, locally-attached disks for tenants with the necessity of supporting operational and management tasks for datacenter operators. The tenant interfaces to virtualized storage, enumerated in Figure 2, closely match that of the underlying hardware with commands such as read, write, and deallocate[2] providing the same semantics as the corresponding NVMe commands.

Device Partitioning: dVols provide tenants a sparse virtual address space backed by an SCM. As the storage requirements of tenants vary over time, the capacity utilization of dVols constantly grows and shrinks during execution. Consequently, Decibel must manage the fine-grained allocation of capacity resources across dVols.

One alternative for device partitioning is to rely on hardware-based NVMe namespaces [71] which divide SCMs into virtual partitions that may be presented directly to tenants. As implemented in modern hardware, namespaces represent large contiguous physical regions of the device, making them unsuitable for dynamically resizing workloads. Moreover, many NVMe devices do not support namespaces at all, and where they are supported, devices are typically limited to a very small number[3] of namespace instances. While the namespace idea is in principle an excellent abstraction at the device layer, these limits make them insufficient today, and are one of the reasons that Decibel elects to virtualize the SCM address space above the device itself.

Decibel virtualizes SCMs at block-granularity. Blocks are 4K contiguous regions of the device's physical address space. While some SCMs support variable block sizes, Decibel uses 4K blocks to match both existing storage system designs and x86 memory pages. Blocks are the smallest writeable unit that do not require firmware read-modify-write (RMW) cycles during updates, and also generally the largest unit that can be atomically overwritten by SCMs for crash-safe in-place updates.

Address Virtualization: dVols map the virtual address space presented to tenants onto physical blocks using a private *virtual-to-physical* (V2P) table. Each dVol's V2P table is structured as a persistent B+ tree, with fixed, block-sized internal and leaf nodes, and is keyed by 64-bit virtual addresses; internal nodes store references to their children as 64-bit physical block addresses.

V2P mappings are stored as metadata on the SCM. Client writes are fully persisted, including both data and V2P mappings, before being acknowledged. Decibel performs soft-update-ordered [19] writes of data blocks and metadata: where a write requires an update to the V2P table, data is always written and acknowledged by the device before the associated metadata write is issued. The current implementation is conservative, in that all V2P transactions are isolated. There is opportunity to further improve performance by merging V2P updates. Subsequent writes to allocated blocks do not modify the V2P table and occur in-place, relying on the block-level write atomicity of SCMs for consistency.

Modern SCMs show little benefit for physically contiguous accesses, especially in multi-tenant scenarios with

[2] We use the NVMe "deallocate" command, also termed "trim", "unmap", or "discard" in the context of SATA/SAS SSDs.

[3] The maximum number supported today is 16, with vendors indicated that devices supporting 128 namespaces are likely to be available over the next few years.

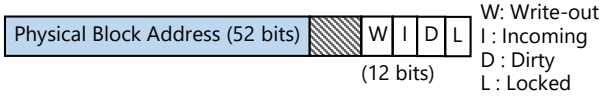


Figure 3: Cached V2P Entry

mixed reads and writes. As a result, dVols do not preserve contiguity from tenant writes and split large, variable-sized requests into multiple, block-sized ones. V2P mappings are also stored at a fixed, block-sized granularity. This trades request amplification and an increase in V2P entries for simplified system design: Decibel does not require background compaction and defragmentation services, while V2P entries avoid additional metadata for variable-sized mappings.

Decibel aggressively caches the mappings for every dVol in DRAM. The V2P table for a fully-allocated terabyte-sized device occupies approximately 6GB (an overhead of 0.6%). While non-trivial, this is well within the limits of a high performance server. Cached V2P entries vary from the on-device format: as Figure 3 illustrates, physical addresses are block-aligned and require only 52 bits, so the remaining 12 bits are used for entry metadata.

The *Incoming* and *Write-out* bits are used for cache management and signify that the entry is either waiting to be loaded from the SCM or that an updated entry is being flushed to the SCM and is awaiting an acknowledgement for the write. The *Dirty* bit indicates that the underlying data block has been modified and is used to track dirtied blocks to copy during dVol migrations. The *Locked* bit provides mutual exclusion between requests to overlapping regions of the dVol: when set, it restricts all access to the mapping for any request context besides the one that has taken ownership of the lock.

Block Allocation: Requests to allocate blocks require a consistent view of allocations across the entire system to prevent races and double allocations. Decibel amortizes the synchronization overhead of allocations by splitting them into *reservations* and *assignment*: each core reserves a fixed-size, physically-contiguous collection of blocks called an *extent* from the device in a single operation and adds it to a per-core allocation pool (resembling the thread cache in `tcmalloc`).

As seen in Figure 4, SCMs are divided into multiple extents, which are dynamically reserved by cores. The reservation of extents to cores is tracked by a global, per-device allocator. Cores asynchronously request more extents from the allocator once the number of available blocks in their local pool falls below a certain threshold. This ensures that, as long as the device is not full, allocations succeed without requiring any synchronization.

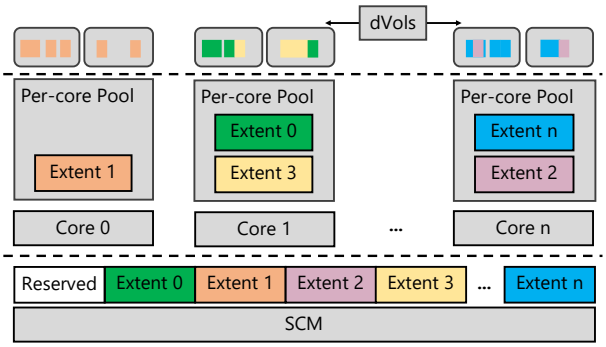


Figure 4: Physical Partitioning of Storage Devices

Assigning entire extents to dVols risks fragmentation and a wastage of space. Instead, cores satisfy allocation requests from dVols by assigning them blocks from any extent in their private pool at a single block granularity. As individual blocks from extents are assigned to different dVols, the split-allocation scheme eliminates both fragmentation and contention along the datapath.

Internally, extents track the allocation status of blocks using a single block-sized bitmap; as every bit in the bitmap represents a block, each extent is 128 MB $(4K \times 4K \times 8)$. Restricting the size of extents to the representation capacity of a single block-sized bitmap allows the bitmap to atomically be overwritten after allocations and frees.

dVols explicitly free blocks that are no longer need using the deallocate command, while deleting dVols or migrating them across devices implicitly triggers block deallocations. Freed blocks are returned to the local pool of the core where they were originally allocated and are used to fulfil subsequent allocation requests.

Data Integrity and Paravirtual Interfaces: SCMs are increasingly prone to block errors and data corruption as they age and approach the endurance limits of flash cells [57]. Even in the absence of hardware failures, SCMs risk data corruption due to *write-tearing*: since most SCMs do not support atomic multi-block updates, failures during these updates result in partial writes that leave blocks in an inconsistent state.

Decibel provides additional services not directly available in hardware to help prevent and detect data corruption. On each write, it calculates block-level checksums and verifies them on reads to detect corrupted blocks before they propagate through the system. dVols also support two additional I/O commands: *multi-block atomicity* to protect against write-tearing and *block compare-and-swap* to allow applications that can only communicate over shared storage to synchronize operations using persistent, on-disk locks [67].

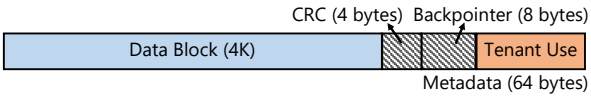


Figure 5: Extended Logical Blocks (Block + Metadata)

Several enterprise SCMs support storing per-block metadata alongside data blocks and updating the metadata atomically with writes to the data. The block and metadata regions together are called *extended logical blocks* (shown in Figure 5). Block metadata corresponds to the Data Integrity Field (DIF) provided by SCSI devices and is intended for use by the storage system. Decibel utilizes this region to store a CRC32-checksum of every block.

Block checksums are self-referential integrity checks that protect against data corruption, but offer no guarantees about metadata integrity, as V2P entries pointing to stale or incorrect data blocks are not detected. Metadata integrity can be ensured either by storing checksums with the metadata or by storing backpointers alongside the data. To avoid updating the mappings on every write, Decibel stores backpointers in the metadata region of a block. As the data, checksum, and backpointer are updated atomically, Decibel overwrites blocks in-place and still remains crash consistent.

The metadata region is exposed to tenants through the extended, metadata-aware read and write commands (see Figure 2) and can be used to store application-specific data, such as version numbers and cryptographic capabilities. As this region is shared between Decibel and tenants, the extended read and write functions mask out the checksum and backpointer before exposing the remainder of the metadata to tenants.

Since most SCMs are unable to guarantee atomicity for writes spanning multiple blocks, Decibel provides atomic updates using block-level copy-on-write semantics. First, new physical blocks are allocated and the data written, following which the corresponding V2P entries are modified to point to the new blocks. Once the updated mappings are persisted, the old blocks are freed. As the V2P entries being updated may span multiple B-tree nodes, a lightweight journal is used to transactionalize the update and ensure crash-consistency.

To perform block-level CAS operations, Decibel first ensures that there are no in-flight requests for the desired block before locking its V2P entry to prevent access until the operation is complete. The entire block is then read into memory and tested against the compare value; if they match, the swap value is written to the block. Storage systems have typically used CAS operations, when available, to co-ordinate accesses to ranges of a shared device or volume without locking the entire device.

vol_create *()* →*(vol, token)*
vol_restrict *(vol, type, param)* →*status*
vol_open *(vol, token)* →*status*
vol_change_auth *(vol, token)* →*newtoken*
vol_delete *(vol, token)* →*status*

Figure 6: Provisioning and Access Control Interfaces

Provisioning and Access Control: Access control mechanisms restrict a tenant's view of storage to only the dVols it is permitted to access. Decibel uses a lightweight, token-based authentication scheme for authorization and *does not* protect data confidentiality via encryption, as such CPU-intensive facilities are best left to either the clients or the hardware.

Figure 6 enumerates the control plane interfaces, presented to tenants, to provision dVols and manage access control policies for them. While creating dVols, Decibel generates a random globally unique identifier and token pair, which are returned to the tenant for use as a volume handle and credentials for future access.

In addition to credential-based authorization, dVols can also restrict access on the basis of network parameters, such as a specific IP address, or to certain VLANs and VXLANs[4]. By forcing all traffic for a dVol onto a private network segment, Decibel allows policies to be applied within the network; for example, traffic can be routed through middleboxes for packet inspection or rely on traffic shaping to prioritize latency-sensitive workloads.

3.2 Scheduling Storage

Virtualization allows tenants to operate as if deployed on private, local storage, while still benefiting from the flexibility and economic benefits of device consolidation within the datacenter. For practical deployments, preserving only an interface resembling local storage is insufficient: the storage system must also preserve the performance of the device and insulate tenants from interference due to resource contention and sharing.

The need for performance isolation in multi-tenant storage systems has led to the development of several algorithms and policies to provide fair sharing of devices and guaranteeing tenant throughput [44, 21, 68, 22, 23, 37, 61, 69, 66, 60, 73] and for providing hard deadlines for requests [37, 22, 69, 73]. Rather than prescribing a particular policy, Decibel provides dVols as a policy enforcement tool for performance isolation.

[4] Virtual Extensible LANs (VXLANs) provide support for L2-over-L4 packet tunneling and are used to build private overlay networks.

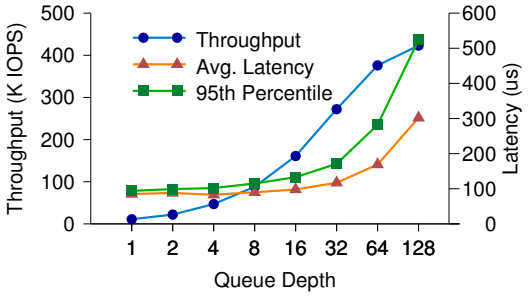


Figure 7: Throughput and Latency of Reads

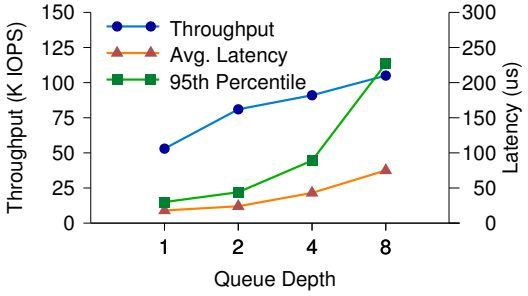


Figure 8: Throughput and Latency of Writes

SLOs: Throughput and Latency: Figures 7 and 8 compare the throughput and latency for both reads and writes of a single device, measured locally at different queue depths. The results lead us to two observations about performance isolation for SCMs:

There is a single latency class for the entire device. Even for multi-queue devices, request latency depends on overall device load. Despite the fact that the NVMe specification details multiple scheduling policies across submission queues for QoS purposes, current devices do not sufficiently insulate requests from different queues to support multiple latency classes for a single device. Instead Decibel allows the storage administrator to pick a throughput target for every SCM, called the *device ceiling*, to match a desired latency target.

Providing hard latency guarantees is not possible on today's devices. Comparing average and 95*th* percentile latencies for the device, even at relatively low utilization levels, reveal significant jitter, particularly in the case of writes. Long tail latencies have also been observed for these devices in real deployments [25]. This is largely due to the Flash Translation Layer (FTL), a hardware indirection layer that provides background bookkeeping operations such as wear levelling and garbage collection.

Emerging hardware that provides predictable performance by managing flash bookkeeping in software is discussed in Section 5. In the absence of predictable SCMs, Decibel focuses on preserving device throughput. dVols encapsulate an SLO describing their performance requirements, either in terms of a proportional share of the device or a precise throughput target.

Characterizing Request Cost: Guaranteeing throughput requires the scheduler to be able to account for the cost of every request before deciding whether to issue it to the device. Request costs, however, are variable and a function of the size and nature of the request, as well as the current load on the SCM. For example, writes are significantly cheaper than reads as long as they are being absorbed by the SCM write buffer, but become much more expensive once the write buffer is exhausted.

The Decibel scheduler does not need to account for variable-sized tenant requests as the address translation layer of the dVol converts them into uniform 4K requests at the block layer. As a simplifying assumption, the scheduler does not try and quantify the relative costs of reads and writes, but instead requires both the device ceiling and SLOs to specify read and write targets separately. In the future, we intend to extend the scheduler to provide a unified cost model for reads and writes [60].

Request Windows: At a specified device ceiling, Decibel determines the size of the global request window for an SCM, or the total number of outstanding requests that can be issued against the device. Each per-core runtime has a private request window, where the sizes of all the individual request windows are equal to that of the global request window for the device. The size of the private request window for cores is calculated on the basis of the SLO requirements of dVols scheduled to execute on that core. As dVols are created, opened, moved, or destroyed, Decibel recalculates the private window sizes, which are periodically fetched by the cores.

dVols submit requests to devices by enqueuing them in private software queues. While submitting requests to the device, the per-core runtime selects requests from the individual dVol queues until either the request window is full or there are no pending requests awaiting submission. The runtime chooses requests from multiple dVols on the basis of several factors, such as the scheduling policy, the dVol's performance requirements, and how many requests the dVols have submitted recently.

Execution Model: Per-core runtimes co-operatively schedule dVols on a single OS thread: the request processor issues asynchronous versions of blocking syscalls and yields in a timely manner. Decibel polls NICs and SCMs on the same thread to eliminate context switching overheads and to allow the schedulers to precisely con-

trol the distribution of compute cycles between servicing the hardware and the request processing within dVols.

Request processing within the dVol includes resolving virtualization mappings and performing access control checks; consequently, requests may block and yield several times during execution and cannot be run to completion as in many memory-backed systems. The scheduler treats requests and dVols as analagous to threads and processes – scheduling operates at the request level on the basis of policies applying to the entire dVol. At any given time the scheduler dynamically selects between executing the network or storage stacks, and processing one of several executable requests from multiple dVols.

Storage workloads are bursty and susceptible to incast [54]; as a result, Decibel is periodically subject to bursts of heavy traffic. At these times, the Decibel scheduler elastically steals cycles to prioritize handling network traffic. It polls NICs at increased frequencies to ensure that packets are not dropped due to insufficient hardware buffering, and processes incoming packets just enough to generate ACKs and prevent retransmissions.

Prioritizing network I/O at the cost of request processing may cause memory pressure due to an increase in number of pending requests. At a certain threshold, Decibel switches back to processing requests, even at the cost of dropped packets. As dropped packets are interpreted as network congestion, they force the sender to back-off, thus inducing back pressure in the system.

Scheduling Policies: The scheduling policy determines how the per-core runtime selects requests from multiple dVols to submit to the device. To demonstrate the policy-agnostic nature of Decibel's architecture, we prototype two different scheduling policies for dVols.

Strict Time Sharing (STS) emulates local storage by statically partitioning and assigning resources to tenants. It sacrifices elasticity and the ability to handle bursts for more predictable request latency. Each dVol is assigned a fixed request quota per scheduling epoch from which the scheduler selects requests to submit to the device. dVols cannot exceed their quota even in the absence of any competition. Further, dVols do not gain any credits during periods of low activity, as unused quota slots are not carried forward.

Deficit Round Robin (DRR) [59] is a work conserving scheduler that supports bursty tenant access patterns. DRR guarantees that each dVol is able to issue its fair share of requests to the device, but does not limit a dVol to only its fair share in the absence of competing dVols. Each dVol has an assigned quota per scheduling epoch; however, dVols that do not use their entire quota carry it forward for future epochs. As dVols can exceed their quota in the absence of competition, bursty workloads can be accommodated.

By default, Decibel is configured with DRR to preserve the flexibility benefits of disaggregated storage. This remains a configurable parameter to allow storage administrators to pick the appropriate policies for their tenants.

3.3 Placement and Discovery

Decibel makes the decision to explicitly decouple *scheduling* from the *placement* of dVols on the appropriate cores and hosts in the cluster. This division of responsibilities allows the scheduler to focus exclusively on, as seen earlier, providing fine-grained performance isolation and predictable performance over microsecond timeframes on a per-core basis. Placement decisions are made with a view of the cluster over longer timeframes in response to the changing capacity and performance requirements of dVols. Decibel defers placement decisions to an external placement engine called Mirador [70]. Mirador is a global controller that provides continuous and dynamic improvements to dVol placements by migrating them across cores, devices, and hosts and plays a role analagous to that of an SDN controller for network flows.

Storage workloads are impacted by more than just the local device; for example, network and PCIe bandwidth oversubscription can significantly impact tenant performance. Dynamic placement with global resource visibility is a response to not just changing tenant requirements, but also to connectivity bottlenecks within the datacenter. Dynamic dVol migrations, however, raise questions about how tenants locate and access their dVols.

dVol Discovery: Decibel implements a global directory service that maps dVol identifiers to the precise host and core on which they run. Cores are independent network-addressable entities with a unique `<ip:port>` identifier and can directly be addressed by tenants. The demultiplexing of tenant requests to the appropriate dVol happens at the core on the basis of the dVol identifier.

dVol Migration: The placement engine triggers migrations in response to capacity or performance shortages and aims to find a placement schedule that ensures that both dVol SLOs are met and that the free capacity of the cluster is uniformly distributed across Decibel instances to allow every dVol an opportunity to grow. Migrations can be across cores on the same host or across devices within the same or different hosts.

Core migrations occur entirely within a single Decibel instance. dVols are migrated to another core within the same host without requiring any data movement. First, Decibel flushes all the device queues and waits for in-flight requests to be completed, but no new dVol requests

are admitted. The dVol metadata is then moved to the destination core and the address directory updated. The client is instructed to invalidate its directory cache and the connection is terminated; the client then connects to the new runtime instance and resumes operations.

Device migrations resemble virtual machine migration and involve a background copy of dVol data. As the dVol continues to service requests, modified data blocks are tracked using the dirty bit in the V2P table and copied to the destination. When both copies approach convergence, the client is redirected to the new destination using the same technique as core migrations and the remaining modified blocks moved in a post-copy pass.

Decibel originally intended to perform migrations without any client involvement using OpenFlow-based redirection in the network and hardware flow steering at the end-hosts. Due to the limited availability of match rules at both switches and NICs today, Decibel opts to defer this functionality to the client library.

3.4 The Network Layer

Decibel presents the dVol interface over asynchronous TCP/IP-based RPC messages. Network flows are processed using a user space networking stack that borrows the TCP state machine and structures for processing TCP flows, such as the socket and flow tables, from mTCP [34] and combines them with custom buffer management and event notification systems. Decibel offloads checksums to the hardware, but currently does not support TCP segmentation offload.

Clients discover core mappings of dVols using the network address directory. As dVols are pinned to cores which have exclusive ownership over them, tenants must direct requests to the appropriate core on the system. Modern NICs provide the ability to precisely match specific fields in the packet header with user-defined predicates and determine the destination queue for that packet on the basis of provided rules. As each core has a unique `<ip:port>`, Decibel uses such flow steering to distribute incoming requests across cores directly in hardware.

For performance reasons, Decibel extends the shared-nothing architecture into the networking layer. It borrows ideas from scalable user space network stacks [56, 34, 45, 5, 52] and partitions the socket and flow tables into local, per-core structures that can be accessed and updated without synchronization.

Memory Management: Decibel pre-allocates large per-core regions of memory for sockets, flow tables, and socket buffers from regular memory and `mbufs` for network packets from hugepages. `mbufs` are stored in lockless, per-socket send and receive ring buffers; the latter

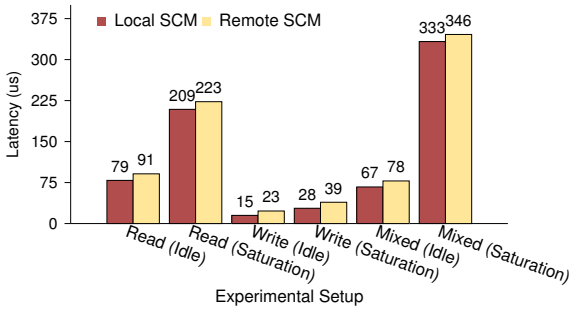


Figure 9: Local and Remote Latency for 4K Requests

is passed to DPDK which uses them to DMA incoming packets. Decibel does not support zero-copy I/O: incoming packet payloads are staged in the receive socket buffer for unpacking by the RPC layer, while writes are buffered in the send socket buffer before transmission. Zero-copy shows little benefit on processors with Direct Data I/O (DDIO), i.e., the ability to DMA directly into cache [45]. Further, once packets in `mbufs` are sent to the DPDK for transmission, they are automatically freed and unavailable for retransmissions, making zero-copy hard without complicating the programming interface.

Event Notifications and Timers: As request processing and the network stack execute on the same thread, notifications are processed in-line via callbacks. Decibel registers callbacks for new connections and for I/O events: `tcp_accept()`, `tcp_rcv_ready()`, and `tcp_snd_ready()`. Send and receive callbacks are analagous to `EPOLLIN` and `EPOLLOUT` and signify the availability of data or the ability to transmit data. The callbacks themselves do not carry any data but are only event notifications for the request processor to act on using the socket layer interfaces. Timers for flow retransmissions and connection timeouts, similarly, cannot rely on external threads or kernel interrupts to fire and instead are tracked using a hashed timer wheel and processed in-line along with other event notifications.

Why not just use RDMA? Several recent high-performance systems have exploited RDMA (Remote Direct Memory Access) – a hardware mechanism that allows direct access to remote memory without software mediation – to eliminate network overheads and construct a low-latency communication channel between servers within a datacenter, in order to accelerate network services, such as key-value stores [48, 16, 35] and data parallel frameworks [33, 24].

RDMA's advantage over traditional networking shrinks as request sizes grow [48, 35], especially in the presence of low-latency, kernel-bypass I/O libraries. Figure 9 compares local and remote access latencies, over

TCP/IP-based messaging, for SCMs when they are relatively idle (for minimum latencies) and at saturation. For a typical storage workload request size of 4K, conventional messaging adds little overhead to local accesses.

RDMA has traditionally been deployed on Infiniband and requires lossless networks for performance, making it hard to incorporate into existing Ethernet deployments. On Ethernet, RDMA requires an external control plane to guarantee packet delivery and ordering [24] and for congestion control to ensure link fairness [74].

Decibel's choice of traditional Ethernet-based messaging is pragmatic, as the advantages of RDMA for storage workloads do not yet outweigh the significant deployment overheads. As RDMA-based deployments increase in popularity, and the control plane protocols for prioritizing traffic and handling congested and lossy networks are refined, this may no longer hold true. Consequently, Decibel's architecture is mostly agnostic to the messaging layer and is capable of switching to RDMA if required by the performance of future SCMs.

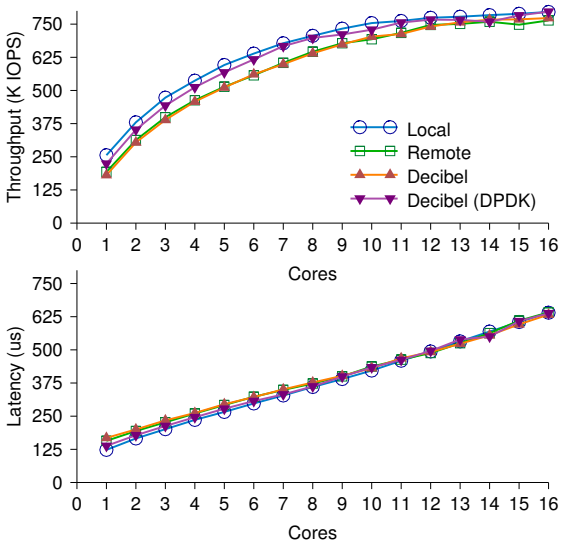


Figure 10: Performance of Decibel for a 70/30 read-write workload. Compared to local storage, Decibel has an overhead of 30µs at device saturation using a DPDK-based client.

4 Evaluation

Decibel is evaluated on a pair of 32-core Haswell systems, each with 2x40GbE X710 NICs and 4x800 GB P3700 NVMe PCIe SSDs, with one system acting as the server and the other hosting multiple clients. Each machine has 64GB RAM split across two NUMA nodes, while the 40GbE interfaces are connected via an Arista 7050 series switch. Both systems run a Linux 4.2 kernel, however, on the server Decibel takes exclusive ownership of both the network and storage adapters. Clients are measured both using the default kernel I/O stack and the DPDK-based network stack from Decibel.

At 4K request sizes, each P3700 is capable of up to 460K random read IOPS, 100K random write IOPS, and 200K random mixed (at a 70/30 read to write ratio) IOPS [31], making the saturated throughput of the system up to 1.8M read IOPS and 800K mixed IOPS. Benchmarking flash-based SSDs is non-trivial as there are a number of factors that may affect their performance. First, the performance of a new SSD is not indicative of how it would perform at *steady state* with fresh drives outperforming their steady state counterparts by a factor or two or three.

Even once steady state is reached, there is a great deal of variability in performance. Transitions from sequential to random and vice versa impact performance for several minutes, while the garbage collector can throttle disk throughput for several seconds. The P3700s, in particular, perform well past their rated write throughput for almost a minute following a period of idleness [55]. The results reported here are the average across a 10 minute run and follow industry standard guidelines for benchmarking [62]: first the devices were pre-conditioned with several weeks of heavy usage and then primed by running the same workload access pattern as the benchmark for 10 minutes prior to the benchmark run.

Remote Overhead and Scalability: Decibel is evaluated for multi-core scalability and to quantify the overhead of disaggregating SCMs when compared to direct-attached storage. All the tests are run against all 4 devices in the system, with clients evenly distributed across the cores. The clients are modelled after `fio` and access blocks randomly across the entire address space. Local clients execute as a single pinned client per-core with a queue depth of 32, while there are 2 remote clients per-core, each operating with a queue depth of 16 requests.

In the *Local* configuration, clients run directly on the server and access raw physical blocks from the SCMs without any virtualization. This local configuration serves as a baseline to compare the overhead of Decibel. In *Remote*, clients run separately from the server and request raw physical blocks across the network over TCP/IP. For *Decibel*, SCMs are virtualized into per-client dVols. Each client has a single dVol that is populated until the SCM is filled, after which they access and update blocks within the dVol. The remote configuration measures pure network overhead when compared to directly-attached SCMs, as well as the overhead of virtualization when compared to Decibel. The *Decibel (DPDK)* configuration is identical to Decibel, except that the clients bypass the kernel and use a DPDK-based network stack.

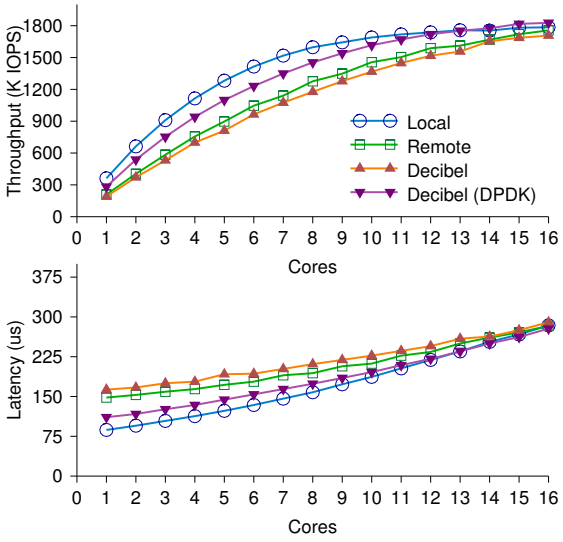


Figure 11: Performance of Decibel for an all reads workload. Compared to local storage, Decibel has an overhead of less than 20µs at device saturation using a DPDK-based client.

Figure 10 compares the performance of all four configurations over 16 cores using a typical storage workload of random mixed 4K requests in a 70/30 read-write ratio. As device saturation is achieved, we do not evaluate Decibel at higher degrees of parallelism.

Decibel is highly scalable and is able to saturate all the devices, while presenting storage across the network with latencies comparable to local storage. DPDK-based clients suffer from an overhead of less than 30µs when compared to local storage, while legacy clients have overheads varying from 30-60µs depending on load.

SCMs offer substantially higher throughput for read-only workloads compared to mixed ones making them more heavily CPU-bound. Figure 11 demonstrates Decibel's ability to saturate all the devices for read-only workloads: the increased CPU load of processing requests is reflected in the gap with the local workload at low core counts. As the number of cores increase, the workload becomes SCM-bound; Decibel scales well and is able to saturate all the devices. At saturation, the DPDK-based client has an overhead of less than 20µs, while legacy clients suffer from overheads of approximately 90µs.

Once the devices are saturated, adding clients increases latency purely due to queueing delays in software. All configurations saturate the devices at less than 16 cores; hence the latency plots in Figures 10 and 11 include queueing delays and do not accurately reflect end-to-end latencies. Table 2 compares latencies at the point of device saturation: for both workloads, Decibel imposes an

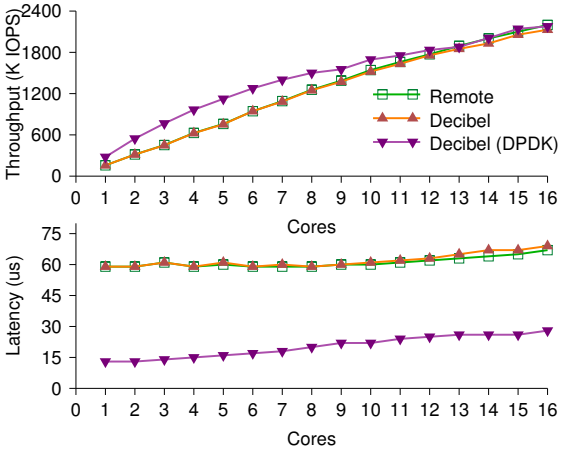


Figure 12: Remote access latencies for Decibel at different degrees of device utilization against DRAM-backed storage.

	70/30		All Reads	
	Xput	Lat	Xput	Lat
Local	750K	422	1.7M	203
Remote	740K	488	1.7M	283
Decibel	740K	490	1.7M	290
Decibel (DPDK)	750K	450	1.7M	221

Table 2: Performance for Workloads (Latency in µs)

overhead of 20-30µs on DPDK-based clients compared to local storage.

Future SCMs, such as 3DXpoint [49], are expected to offer sub-microsecond latencies for persistent memories and approximately 10µs latencies for NVMe storage. With a view towards these devices, we evaluate Decibel against a DRAM-backed block device. As seen in Figure 12, DPDK-based clients have remote access latencies of 12-15µs at moderate load, which increases to 26µs at NIC saturation. Legacy clients have access latencies higher than 60µs, which demonstrates that the kernel stack is a poor fit for rack-scale storage architectures.

dVol Isolation: Performance isolation in Decibel is evaluated by demonstrating fair sharing of a device in two different scheduling policies when compared with a First-Come, First-Served (FCFS) scheduler that provides no performance isolation. Strict Timesharing (STS) provides static resource partitioning, while in Deficit Round Robin (DRR), dVols are prevented from interfering with the performance of other dVols, but are allowed to consume excess, unused bandwidth.

To illustrate performance isolation, Decibel is evaluated with three clients, each with a 70/30 mixed random

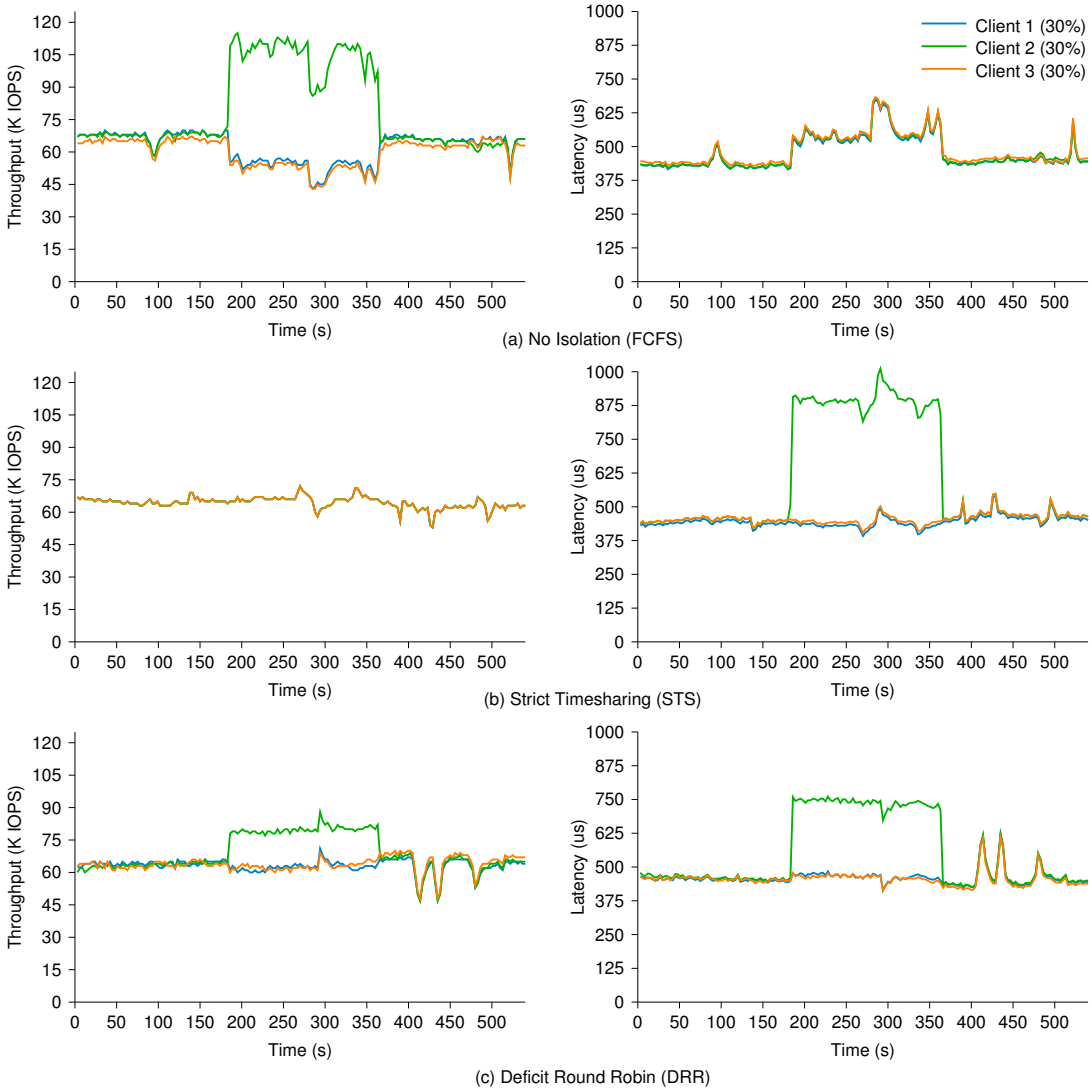


Figure 13: Isolation of a single device across multiple workloads in Decibel. Compared to the no isolation case in (a), the scheduling policies in (b) and (c), provide clients 1 and 3 a fair share of the device, even in the face of the bursty accesses of client 2.

workload, against a single shared SCM. Each client continuously issues requests to its own dVol, such that the dVol has 30 outstanding requests at any time. The dVols are configured to have an equal proportion, i.e., 30%, of the total device throughput, while the device ceiling is set to 100% utilization.

As seen in Figure 13, each client receives a throughput of 60K, leaving the device at 90% saturation. At the 3 minute mark, one of the clients experiences a traffic burst for 3 minutes such that it has 90 simultaneous in-flight requests. At 6 minutes, the burst subsides and the client returns to its original load.

FCFS offers no performance isolation, allowing the burst to create queueing overheads which impact throughput and latency of all other clients by 25%. After the burst subsides, performance returns to its original level. In contrast, STS preserves the throughput of all the clients and prevents clients from issuing any requests beyond their 30% reservation. As each dVol has hard reservations on the number of requests it can issue, requests from the bursty client are queued in software and see huge spikes in latency. The performance of the other clients remains unaffected at all times, but the excess capacity of the device remains unutilized.

DRR both guarantees the throughput of other clients and is work conserving: the bursty client consumes the unused bandwidth until the device ceiling is reached, but

not at the cost of the throughput of other clients. Latency, however, for all the clients rises slightly – this is not because of queueing delays, but because the device latency increases as it gets closer to saturation.

5 Related Work

Network-Attached Storage: The idea of centralizing storage in consolidated arrays and exporting disks over the network [38] is not a new one and has periodically been explored with changes in the relative performance of CPU, networks, and disks. For spinning disk based systems, Petal [41] is a virtualized block store that acts as a building block for a distributed file system [65]. While Petal focuses on aggregating physical disks for performance, Decibel is concerned with the performance challenges of building isolated volumes for SCMs.

More recently, network-attached storage for flash devices has been used in Corfu [4] and Strata [15]. Corfu presents a distributed log over virtualized flash block-devices while storing address translations at the clients. As the clients are trusted, co-operating entities, Corfu does not attempt to provide isolation between them. Strata focuses on providing a global address space for a scalable network file system on top of network-attached storage, and discusses the challenges in providing data plane services such as device aggregation, fault tolerance, and skew mitigation in a distributed manner. In contrast, Decibel is an example of the high-performance network-attached storage such file systems rely on, and provides the services required for multi-tenancy that cannot safely be implemented higher up the stack.

Network-attached Secure Disks (NASD) [20] explore security primitives and capabilities to allow sharing storage devices without requiring security checks at an external file manager on every request, while Snapdragon [1] uses self-describing capabilities to verify requests and limit the blocks a remote client has access to. SNAD [46] performs both tenant authentication and block encryption at the storage server to restrict unauthorized accesses.

Partitioned Data Stores: VoltDB [63] and MICA [43] are both examples of shared-nothing in-memory data stores, which explore vertical partitioning of hardware resources to allow all operations to proceed without expensive cross-core coordination. Architecturally, the per-core runtimes in Decibel resemble those in these systems with the addition of persistent storage devices and the associated datapath services.

Chronos [36] is a more general framework for partitioning applications by running several independent instances in parallel, fronted by a load balancer aware of the instance to partitioning mapping that can route requests accordingly.

Application Managed Flash: Several recent research storage systems have proposed using open-channel SSDs for more predictable performance [42, 7, 29, 58]. These devices expose internal flash channels, dies, and planes to the system and allow for application-managed software FTLs and custom bookkeeping policies. Of these systems, Flashblox has demonstrated that providing strong isolation and supporting multiple latency classes on a shared SCM requires extending full system partitioning to within the device. By binding device channels and dies directly to tenants in hardware and providing per-tenant accounting for garbage collection, it removes multiple sources of performance interference and maintains low tail latencies in the face of competing tenants.

Application-managed flash is largely complementary to Decibel and focuses largely on providing better and more flexible implementations of services currently provided by the FTL. These systems intentionally maintain a familiar block-like presentation for convenience and, as such, Decibel could integrate with such systems to provide strong end-to-end performance isolation.

6 Conclusion

SCMs represent orders of magnitude changes to the throughput, latency, and density of datacenter storage, and have caused a reconsideration in how storage is presented, managed, and accessed within modern datacenters. Decibel responds to the performance realities of SCMs by providing dVols to delegate storage to tenants within fully disaggregated storage architectures. dVols focus exclusively on virtualizing storage and isolating multiple tenants while ensuring that the storage is accompanied with a committed amount of compute and network resources to provide tenants with predictable, low-latency access to data.

7 Acknowledgments

We would like to thank our shepherd, Anirudh Badam, and the anonymous reviewers for their insightful comments and valuable feedback. Decibel has drawn on the experiences of Coho Data's engineering team in building a high-performance enterprise storage system and has especially benefited from numerous discussions with both Steven Smith and Tim Deegan. Finally, we would like to thank Dave Cohen at Intel for frequently providing a valuable "full-stack" perspective on datacenter storage.

References

[1] AGUILERA, M. K., JI, M., LILLIBRIDGE, M., MACCORMICK, J., OERTLI, E., ANDERSEN, D., BURROWS, M., MANN, T., AND THEKKATH, C. A. Block-Level Security for Network-Attached Disks. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (2003), FAST'03.

[2] AGUILERA, M. K., LENERS, J. B., AND WALFISH, M. Yesquel: Scalable SQL Storage for Web Applications. In *Proceedings of the 25th ACM SIGOPS Symposium on Operating Systems Principles* (2015), SOSP'15.

[3] AVAGO. ExpressFabric: Thinking Outside the Box. `http://www.avagotech.com/applications/datacenters/expressfabric`.

[4] BALAKRISHNAN, M., MALKHI, D., PRABHAKARAN, V., WOBBER, T., WEI, M., AND DAVIS, J. D. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (2012), NSDI'12.

[5] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14.

[6] BJØRLING, M., AXBOE, J., NELLANS, D., AND BONNET, P. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *Proceedings of the 6th International Systems and Storage Conference* (2013), SYSTOR'13.

[7] BJØRLING, M., GONZALEZ, J., AND BONNET, P. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies* (2017), FAST'17.

[8] BOLOSKY, W. J., AND SCOTT, M. L. False Sharing and Its Effect on Shared Memory Performance. In *Proceedings of the 4th USENIX Symposium on Experiences with Distributed and Multiprocessor Systems* (1993), SEDMS'93.

[9] CALDER, B., WANG, J., OGUS, A., NILAKANTAN, N., SKJOLSVOLD, A., MCKELVIE, S., XU, Y., SRIVASTAV, S., WU, J., SIMITCI, H., HARIDAS, J., UDDARAJU, C., KHATRI, H., EDWARDS, A., BEDEKAR, V., MAINALI, S., ABBASI, R., AGARWAL, A., HAQ, M. F. U., HAQ, M. I. U., BHARDWAJ, D., DAYANAND, S., ADUSUMILLI, A., MCNETT, M., SANKARAN, S., MANIVANNAN, K., AND RIGAS, L. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), SOSP'11.

[10] CANTRILL, B., AND BONWICK, J. Real-World Concurrency. *Queue 6*, 5 (Sept. 2008).

[11] CAULFIELD, A. M., MOLLOV, T. I., EISNER, L. A., DE, A., COBURN, J., AND SWANSON, S. Providing Safe, User Space Access to Fast, Solid State Disks. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS'12.

[12] CAULFIELD, A. M., AND SWANSON, S. QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (2013), ISCA'13.

[13] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles* (2009), SOSP'09.

[14] CONFLUENT. Kafka Production Deployment. `http://docs.confluent.io/2.0.1/kafka/deployment.html`, 2015.

[15] CULLY, B., WIRES, J., MEYER, D., JAMIESON, K., FRASER, K., DEEGAN, T., STODDEN, D., LEFEBVRE, G., FERSTAY, D., AND WARFIELD, A. Strata: Scalable High-performance Storage on Virtualized Non-volatile Memory. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (2014), FAST'14.

[16] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), NSDI'14.

[17] EMC. DSSD D5. `https://www.emc.com/en-us/storage/flash/dssd/dssd-d5/index.htm`, 2016.

[18] FORD, D., LABELLE, F., POPOVICI, F. I., STOKELY, M., TRUONG, V.-A., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in Globally Distributed Storage Systems. In *Pro-*

ceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (2010), OSDI'10.

[19] GANGER, G. R., MCKUSICK, M. K., SOULES, C. A. N., AND PATT, Y. N. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Trans. Comput. Syst. 18*, 2 (May 2000), 127–153.

[20] GIBSON, G. A., NAGLE, D. F., AMIRI, K., BUTLER, J., CHANG, F. W., GOBIOFF, H., HARDIN, C., RIEDEL, E., ROCHBERG, D., AND ZELENKA, J. A Cost-effective, High-bandwidth Storage Architecture. In *Proceedings of the 8th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (1998), ASPLOS'98.

[21] GULATI, A., AHMAD, I., AND WALDSPURGER, C. A. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *Proccedings of the 7th USENIX Conference on File and Storage Technologies* (2009), FAST'09.

[22] GULATI, A., MERCHANT, A., AND VARMAN, P. J. pClock: An Arrival Curve Based Approach for QoS Guarantees in Shared Storage Systems. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2007), SIGMETRICS'07.

[23] GULATI, A., MERCHANT, A., AND VARMAN, P. J. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), OSDI'10.

[24] GUO, C., WU, H., DENG, Z., SONI, G., YE, J., PADHYE, J., AND LIPSHTEYN, M. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), SIGCOMM'16.

[25] HAO, M., SOUNDARARAJAN, G., KENCHAMMANA-HOSEKOTE, D., CHIEN, A. A., AND GUNAWI, H. S. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies* (2016), FAST'16.

[26] HITZ, D., LAU, J., AND MALCOLM, M. File System Design for an NFS File Server Appliance. In *Proceedings of the 1994 USENIX Winter Technical Conference* (1994).

[27] HOEFLER, T., ROSS, R. B., AND ROSCOE, T. Distributing the Data Plane for Remote Storage Access. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (2015), HotOS'15.

[28] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., AND YEKHANIN, S. Erasure Coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Annual Technical Conference* (2012), ATC'12.

[29] HUANG, J., BADAM, A., CAULFIELD, L., NATH, S., SENGUPTA, S., SHARMA, B., AND QURESHI, M. K. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies* (2017), FAST'17.

[30] INTEL. Data Plane Development Kit. `http://dpdk.org`, 2012.

[31] INTEL. Intel Solid State Drive DC P3700 Series. `http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-dc-p3700-spec.pdf`, October 2015.

[32] INTEL. Storage Plane Development Kit. `https://01.org/spdk`, 2015.

[33] ISLAM, N. S., RAHMAN, M. W., JOSE, J., RAJACHANDRASEKAR, R., WANG, H., SUBRAMONI, H., MURTHY, C., AND PANDA, D. K. High Performance RDMA-based Design of HDFS over InfiniBand. In *Proceedings of the IEEE International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), SC'12.

[34] JEONG, E. Y., WOO, S., JAMSHED, M., JEONG, H., IHM, S., HAN, D., AND PARK, K. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), NSDI'14.

[35] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM SIGCOMM Conference* (2014), SIGCOMM'14.

[36] KAPOOR, R., PORTER, G., TEWARI, M., VOELKER, G. M., AND VAHDAT, A. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the 3rd ACM Symposium on Cloud Computing* (2012), SoCC'12.

[37] KARLSSON, M., KARAMANOLIS, C., AND ZHU, X. Triage: Performance Differentiation for Storage Systems Using Adaptive Control. *ACM Transactions on Storage (TOS) 1*, 4 (Nov. 2005).

[38] KATZ, R. H. High Performance Network and Channel-Based Storage. Tech. Rep. UCB/CSD-91-650, EECS Department, University of California, Berkeley, Sep 1991.

[39] KLIMOVIC, A., KOZYRAKIS, C., THERESKA, E., JOHN, B., AND KUMAR, S. Flash storage disaggregation. In *Proceedings of the 11th ACM SIGOPS European Conference on Computer Systems* (2016), EuroSys'16.

[40] KUNZ, G. Impact of Shared Storage on Apache Cassandra. `http://www.datastax.com/dev/blog/impact-of-shared-storage-on-apache-cassandra`, January 2017.

[41] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed Virtual Disks. In *Proceedings of the 7th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (1996), ASPLOS'96.

[42] LEE, S., LIU, M., JUN, S., XU, S., KIM, J., AND ARVIND, A. Application-managed Flash. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies* (2016), FAST'16, pp. 339–353.

[43] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), NSDI'14.

[44] LUMB, C. R., MERCHANT, A., AND ALVAREZ, G. A. Facade: Virtual Storage Devices with Performance Guarantees. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (2003), FAST'03.

[45] MARINOS, I., WATSON, R. N., AND HANDLEY, M. Network Stack Specialization for Performance. In *Proceedings of the 2014 ACM SIGCOMM Conference* (2014), SIGCOMM'14.

[46] MILLER, E. L., LONG, D. D. E., FREEMAN, W. E., AND REED, B. Strong Security for Network-Attached Storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002), FAST'02.

[47] MILLER, J. Why does my choice of storage matter with Cassandra? `http://www.slideshare.net/johnny15676/why-does-my-choiceofstorage-matterwithcassandra`, November 2014.

[48] MITCHELL, C., GENG, Y., AND LI, J. Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store. In *Proceedings of the 2013 USENIX Annual Technical Conference* (2013), ATC'13.

[49] MORGAN, T. P. Intel Shows Off 3D XPoint Memory Performance. `http://www.nextplatform.com/2015/10/28/intel-shows-off-3d-xpoint-memory-performance/`, October 2015.

[50] OUYANG, J., LIN, S., JIANG, S., HOU, Z., WANG, Y., AND WANG, Y. SDF: Software-defined Flash for Web-scale Internet Storage Systems. In *Proceedings of the 19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), ASPLOS'14.

[51] PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (1988), SIGMOD'88.

[52] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14.

[53] PETERSEN, C. Introducing Lightning: A flexible NVMe JBOF. `https://code.facebook.com/posts/989638804458007/introducing-lightning-a-flexible-nvme-jbof/`, March 2016.

[54] PHANISHAYEE, A., KREVAT, E., VASUDEVAN, V., ANDERSEN, D. G., GANGER, G. R., GIBSON, G. A., AND SESHAN, S. Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008), FAST'08.

[55] RILEY, D. Intel SSD DC P3700 800GB and 1.6TB Review: The Future of Storage. `http://www.tomshardware.com/reviews/intel-ssd-dc-p3700-nvme,3858-5.html`, August 2014.

[56] RIZZO, L. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Annual Technical Conference* (2012), ATC'12.

[57] SCHROEDER, B., LAGISETTY, R., AND MERCHANT, A. Flash Reliability in Production: The Expected and the Unexpected. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies* (2016), FAST'16.

[58] SHEN, Z., CHEN, F., JIA, Y., AND SHAO, Z. DIDACache: A Deep Integration of Device and Application for Flash-Based Key-Value Caching. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies* (2017), FAST'17.

[59] SHREEDHAR, M., AND VARGHESE, G. Efficient Fair Queueing Using Deficit Round Robin. In *Proceedings of the 1995 ACM SIGCOMM Conference* (1995), SIGCOMM'95.

[60] SHUE, D., AND FREEDMAN, M. J. From Application Requests to Virtual IOPs: Provisioned Key-value Storage with Libra. In *Proceedings of the 9th ACM SIGOPS European Conference on Computer Systems* (2014), EuroSys'14.

[61] SHUE, D., FREEDMAN, M. J., AND SHAIKH, A. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (2012), OSDI'12.

[62] SPANJER, E., AND HO, E. The Why and How of SSD Performance Benchmarking. `http://www.snia.org/sites/default/education/tutorials/2011/fall/SolidState/EstherSpanjer_The_Why_How_SSD_Performance_Benchmarking.pdf`, 2011.

[63] STONEBRAKER, M., MADDEN, S., ABADI, D. J., HARIZOPOULOS, S., HACHEM, N., AND HELLAND, P. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases* (2007), VLDB'07, pp. 1150–1160.

[64] SWANSON, S., AND CAULFIELD, A. Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage. *Computer 46*, 8 (Aug. 2013).

[65] THEKKATH, C. A., MANN, T., AND LEE, E. K. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles* (1997), SOSP'97.

[66] THERESKA, E., BALLANI, H., O'SHEA, G., KARAGIANNIS, T., ROWSTRON, A., TALPEY, T., BLACK, R., AND ZHU, T. IOFlow: A Software-defined Storage Architecture. In *Proceedings of the 24th ACM SIGOPS Symposium on Operating Systems Principles* (2013), SOSP'13.

[67] VMWARE. VMware vSphere Storage APIs Array Integration (VAAI). `http://www.vmware.com/resources/techresources/10337`, December 2012.

[68] WACHS, M., ABD-EL-MALEK, M., THERESKA, E., AND GANGER, G. R. Argon: Performance Insulation for Shared Storage Servers. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies* (2007), FAST'07.

[69] WANG, A., VENKATARAMAN, S., ALSPAUGH, S., KATZ, R., AND STOICA, I. Cake: Enabling High-level SLOs on Shared Storage Systems. In *Proceedings of the 3rd ACM Symposium on Cloud Computing* (2012), SoCC'12.

[70] WIRES, J., AND WARFIELD, A. Mirador: An Active Control Plane for Datacenter Storage. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies* (2017), FAST'17.

[71] WORKGROUP, N. E. NVM Express revision 1.2 Specification. `http://nvmexpress.org/wp-content/uploads/NVM_Express_1_2_Gold_20141209.pdf`, November 2014.

[72] YANG, J., MINTURN, D. B., AND HADY, F. When Poll is Better Than Interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012), FAST'12.

[73] ZHU, T., TUMANOV, A., KOZUCH, M. A., HARCHOL-BALTER, M., AND GANGER, G. R. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *Proceedings of the 5th ACM Symposium on Cloud Computing* (2014), SOCC'14.

[74] ZHU, Y., ERAN, H., FIRESTONE, D., GUO, C., LIPSHTEYN, M., LIRON, Y., PADHYE, J., RAINDEL, S., YAHIA, M. H., AND ZHANG, M. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM SIGCOMM Conference* (2015), SIGCOMM'15.

vCorfu: A Cloud-Scale Object Store on a Shared Log

Michael Wei[★†], Amy Tai[◇†], Christopher J. Rossbach[■†], Ittai Abraham[†], Maithem Munshed[‡], Medhavi Dhawan[‡], Jim Stabile[‡], Udi Wieder[†], Scott Fritchie[†], Steven Swanson[★], Michael J. Freedman[◇], Dahlia Malkhi[†]

[†]VMware Research Group, [‡]VMware,
[★]University of California, San Diego, [◇]Princeton University, [■]UT Austin

Abstract

This paper presents vCorfu, a strongly consistent cloud-scale object store built over a shared log. vCorfu augments the traditional replication scheme of a shared log to provide fast reads and leverages a new technique, *composable state machine replication*, to compose large state machines from smaller ones, enabling the use of state machine replication to be used to efficiently in huge data stores. We show that vCorfu outperforms Cassandra, a popular state-of-the art NoSQL stores while providing strong consistency (*opacity*, *read-own-writes*), efficient transactions, and global snapshots at cloud scale.

1 Introduction

Most data stores make a trade-off between *scalability*, or the ability of a system to be resized to meet the demands of a workload and *consistency*, which requires that operations on a system return predictable results. The proliferation of cloud services has led developers to insist on scalable data stores. To meet that demand, a new class of data stores known as NoSQL emerged which partition data, favoring scalability over strong consistency guarantees. While partitioning enables NoSQL stores operate at cloud-scale, it makes operations that are simple in traditional data stores (e.g. modifying multiple items atomically) difficult if not impossible in NoSQL stores.

Systems based on distributed shared logs [9, 10, 11, 40, 41] can address the scalability–consistency tradeoff. Instead of partitioning based on data contents as NoSQL stores do, these systems employ state machine replication (SMR) [27] and achieve scale-out by partitioning based on the order of updates. Since the log provides a single source of ground truth for ordering, shared logs offer a number of attractive properties such as strong consistency and global snapshots.

Shared logs, however, are not without drawbacks. In contrast to NoSQL, clients cannot simply "read" the latest data: the log only stores updates. Instead, clients *play* the log, reading and processing the log sequentially to update their own in-memory views. Playback can easily become a bottleneck: a client may process many updates which are irrelevant to the servicing of a request, dramatically increasing latencies when the system is under load. Figure 1 shows an example in which a client interested in reading Alice's last login time must read updates to other users to find and retrieve the most recent login. As a result, many shared log systems either target metadata services [10] with minimal state and client load, or delegate playback to an intermediate server [9, 11], further increasing latency.

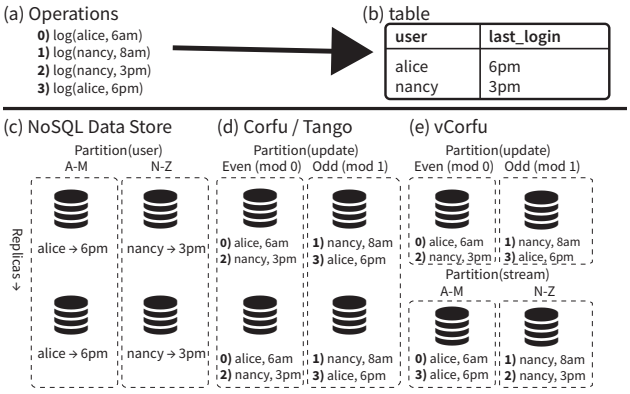


Figure 1: Physical layout of (a) operations on a (b) table stored in a (c) NoSQL data store (d) Shared log systems [9, 10] and (e) vCorfu.

This paper presents vCorfu, which makes distributed, shared log based systems applicable to a much broader array of settings by combining the consistency benefits of a shared log like Corfu [9] and Tango [10] with the locality advantages of scattered logs like Kafka [26] and Kinesis [30]. vCorfu is a cloud-scale, distributed object store. At the core of vCorfu is a scalable, virtualized shared log. The key innovation in vCorfu's log is *materialization*, which divides a single log into virtual logs called

materialized streams. Unlike *streams* proposed in previous systems [10], which support only sequential reads, materialized streams support fast, fully random reads, because all updates for a stream can be accessed from a single partition. This design enables log replicas to use SMR to service requests directly, relieving the burden of playback from clients. Like other shared log systems, vCorfu supports strong consistency, linearizable reads and transactions, but the locality advantages of materialization enable vCorfu to scale to thousands of clients. vCorfu also leverages a sequencer to implement a fast, lightweight transaction manager and can execute read-only transactions without introducing conflicts. A novel technique called *composable state machine replication* (CSMR) enables vCorfu to store huge objects while still allowing client queries to be expressed using a familiar object-based model.

We make the following contributions:

- We present the design and architecture of vCorfu, a cloud-scale distributed object store built on a shared log. vCorfu's novel materialization technique enables reads without playback while maintaining strong consistency and high availability.
- We show that vCorfu's innovative design provides the same strong consistency guarantees as shared log designs while enabling scalability and performance that is competitive with, and often better than current NOSQL systems.
- We demonstrate that by conditionally issuing tokens, our sequencer performs lightweight transaction resolution, relieving clients of the burden of resolving transactions.
- We evaluate vCorfu against a popular NOSQL store, Cassandra, and show that vCorfu is just as fast for writes and much faster at reads, even while providing stronger consistency guarantees and advanced features such as transactions.
- We describe CSMR, a technique which enables efficient storage of huge objects by composition of a large state machine from smaller component state machines. vCorfu can store and support operations against 10GB YCSB! [16] database without sacrificing the strong consistency afforded by SMR.

2 Background

2.1 Data Stores

Modern web applications rely heavily on multi-tiered architecture to enable systems in which components may be scaled or upgraded independently. Traditional architectures consist of three layers: a front-end which communicates to users, an application tier with stateless logic, and a data tier, where state is held. This organization enabled early web applications to scale easily because stateless front-end and application tiers enable scaling *horizontally* in the application tier with the addition of more application servers or *vertically* in the data tier by upgrading to more powerful database servers.

As more and more applications move to cloud execution environments, system and application designers face increasingly daunting scalability requirements in the common case. At the same time, the end of Dennard scaling [21] leaves system builders unable to rely on performance improvements from the hardware: vertical scaling at the data tier is no longer feasible in most settings. As a consequence, modern cloud-scale systems generally trade off reduced functionality and programmability for scalability and performance at the data tier. A new class of NOSQL data stores [1, 4, 12, 14, 18, 26] has emerged, which achieve cloud-scale by relaxing consistency, eliding transaction support, and restricting query and programming models.

A severe consequence of this trend is an increased burden on programmers. In practice, programmers of modern cloud systems are forced to cobble together tools and components to restore missing functionality when it is needed. For example, a lock server such as ZooKeeper [23] is often used in conjunction with a NOSQL store to implement atomic operations. Programmers commonly implement auxiliary indexes to support queries, typically with relaxed consistency since the auxilliary index is not maintained by the data store.

2.2 Scalable Shared Logs

Shared logs have been used to provide highly fault-tolerant distributed data stores since the 1980s [36, 38]. Logs are an extremely powerful tool for building strongly consistent systems, since data is never overwritten, only appended, which yields a total order over concurrent modifications to the log. Early shared logs had limited scalability, as all appends must be serialized through a single server, quickly becoming an I/O bottleneck.

More recent shared log designs [9, 10, 11, 40, 41] address this scalability limitation to varying degrees. For example, the Corfu protocol [9] leverages a centralized sequencer which is not part of the I/O path, yielding a design in which append throughput is only limited by the speed in which a sequencer can issue log addresses.

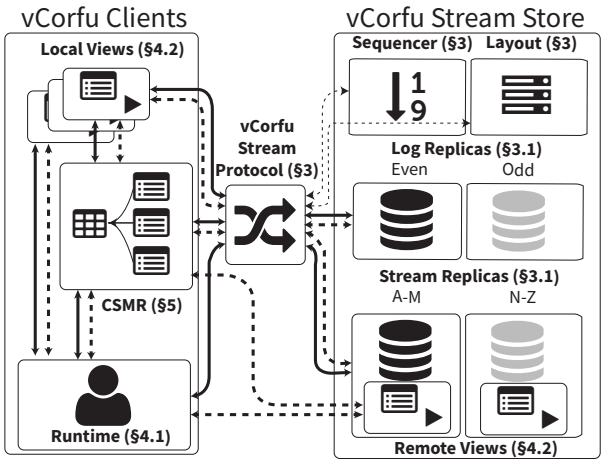


Figure 2: The architecture of vCorfu. Solid lines highlight the write path, while dotted lines highlight the read path. Thin lines indicate control operations outside of the I/O path.

2.3 State Machine Replication

Most shared log systems use state machine replication (SMR) [37] which relies on the log's total ordering of appends to implement an abstraction layer over the log. Data stored on the log is modeled as a state machine. Clients modify data by appending updates to the log and read data by traversing the log and applying those updates in order to an in-memory view. This approach enables strong consistency, and significantly simplifies support for transactions over multiple data items [10, 11].

The achilles' heel of shared log systems, however, is playback. To service *any* request, a client must read every single update and apply it to in-memory state, regardless of whether the request has any dependency on those updates. In practice, this has limited the applicability of shared log systems to settings characterized by few clients or small global state [9], such as metadata services [10, 40]. In contrast, data tiers in typical web applications manage state at a scale that may make traditional playback prohibitively expensive. Worse, in systems relying on stateless application tiers, naïve use of shared logs induces a playback requirement to reconstruct state for every request. The goal of vCorfu is to eliminate these limitations, enabling SMR with shared logs over large state and without client playback overheads.

3 vCorfu Stream Store

vCorfu implements a shared log abstraction that removes the overhead and limitations of shared logs, enabling playback that does not force a client to playback potentially irrelevant updates. vCorfu virtualizes the log using a novel technique called *stream materialization*. Unlike streams in Tango, which are merely tags within a shared log, *materialized streams* are a first class abstraction which supports random and bulk reads just like scattered logs like Kafka [26] and Kinesis [30], but with all the consistency benefits of a shared log like Corfu [9] and Tango [10].

Operation	Description
`read(laddresses)`	Get the data stored at *log address(es)*.
`read(stream, saddresses)`	Read from a *stream* at *stream address(es)*.
`append(stream, data)`	Append *data* to *stream*.
`check(stream)`	Get the last address issued to a *stream*.
`trim(stream, saddresses, prefix)`	Release all entries with *stream address* $< prefix$.
`fillhole(laddress)`	Invoke hole-filling for *log address*.

Table 1: Core operations supported by the vCorfu shared log.

The vCorfu stream store architecture is shown in Figure 2. In vCorfu, data are written to materialized streams, and data entries receive monotonically increasing tokens on both a global log and on individual streams from a *sequencer* server. The sequencer can issue tokens *conditionally* to enable fast optimistic transaction resolution, as described in Section 4. vCorfu writes data in the form of updates to both *log replicas* and *stream replicas*, each of which are indexed differently. This design replicates data for durability, but enables access to that data with different keys, similar to Replex [39]. The advantage is that clients can directly read the latest version of a stream simply by contacting the stream replica.

A *layout* service maintains the mapping from log and stream addresses to replicas. Log replicas and stream replicas in vCorfu contain different sets of updates, as shown in Figure 1. The log replicas store updates by their (global) log address, and stream replicas by their stream addresses. The replication protocol in vCorfu dynamically builds replication chains based on the global log offset, the streams which are written to, and the streams offsets. Subsequent sections consider the design and implementation of materialized streams in more detail.

vCorfu is elastic and scalable: replicas may be added or removed from the system at any time. The sequencer, because it merely issues tokens, does not become an I/O bottleneck. Reconfiguration is triggered simply by changing the active layout. Finally, vCorfu is *fault tolerant* - data which is stored in vCorfu can tolerate a limited number of failures based on the arrangement and number of replicas in the system, and recovery is handled similar to the mechanism in Replex [39]. Generally, vCorfu can tolerate the failures as long as a log replica and stream replica do not fail simultaneously. Stream replicas can be reconstructed from the aggregate of the log replicas, and log replicas can be reconstructed by scanning through all stream replicas.

Operationally, stream materialization divides a single

```
"sequencers": 10.0.0.1,
"segments": {
 "start" : 0,
 "log" : [[ 10.0.1.1 ], [ 10.0.1.2 ]],
 "stream" : [[ 10.0.2.1 ], [ 10.0.2.2 ]] ] }
```

Figure 3: An example layout. Updates are partitioned by their stream id and the log offset; a simple partitioning function mods these values with respect to the number of replicas. An update to stream 0 at log address 1 would be written to 10.0.1.2 and 10.0.2.1, while an update to stream 1 at log address 3 would be written to 10.0.1.2 and 10.0.2.2.

global log into materialized streams, which support logging operations: append, random and bulk reads, trim, check and fillhole; the full API is shown in Table 1. Each materialized stream maps to an *object* in vCorfu, and each stream stores an ordered history of modifications to that object, following the SMR [37] paradigm.

3.1 Fully Elastic Layout

In vCorfu, a mapping called a layout describes how offsets in the global log or in a given materialized stream map to replicas. A vCorfu client runtime must obtain a copy of the most current layout to determine which replica(s) to interact with. Each layout is stamped with an *epoch* number. Replicas will reject requests from clients with a stale epoch. A Paxos-based protocol [27] ensures that all replicas agree on the current layout. An example layout is shown in Figure 3. Layouts work like leases on the log: a client request with the wrong layout (and wrong epoch number) will be rejected by replicas. The layout enables clients to safely contact a stream replica directly for the latest update to a stream.

3.2 Appending to vCorfu materialized streams

A client appending to a materialized stream (or streams) first obtains the current layout and makes a request to the sequencer with a *stream id*. The sequencer returns both a *log token*, which is a pointer to the next address in the global log, and a *stream token*, which is a pointer to the next address in the stream. Using these tokens and the layout, the client determines the set of replicas to write to.

In contrast to traditional designs, replica sets in vCorfu are dynamically arranged during appends. For fault tolerance, each entry is replicated on two replica types: the first indexed by the address in the log (the *log replica*), and the second by the combination of the stream id and the stream address (the *stream replica*). To perform a write, the client writes to the log replica first, then to the stream replica. If a replica previously accepted a write to a given address, the write is rejected and the client must retry with a new log token. Once the client

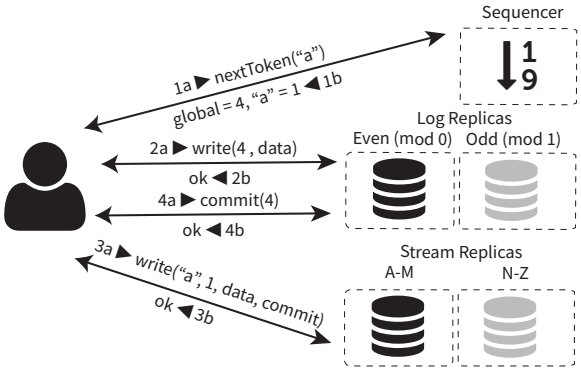


Figure 4: Normal write path of a vCorfu log write, which takes four roundtrips: one for token acquisition, two for writing to each replica (and committing at the stream replica), and one to send a commit message to the log replica.

writes to both replicas, it commits the write by broadcasting a commit message to each replica it accessed (except the final replica, since the final write is already committed). Replicas will only serve reads for committed data. This enables stream replicas to provide a dense materialized stream, without holes. The write path of a client, which takes four roundtrips in normal operation is shown in Figure 4. A server-driven variant where the log replica writes to the stream replica takes 6 messages; we leave implementation of this variant for future work.

3.3 Atomically appending to multiple streams

The primary benefit of materialized streams is that they provide an abstraction of independent logs while maintaining a total global order over all appends. This enables vCorfu to support atomic writes across streams, which form the basic building block for supporting transactions.

To append to multiple streams atomically, the client obtains a log token and stream tokens for each stream it wishes to append to. The client first writes to the log replica using the log token. Then, the client writes to the stream replica of each stream (multiple streams mapped to the same replica are written together so each replica is visited only once). The client then sends a commit message to each participating replica (the commit and write are combined for the last replica in the chain). The resulting write is ordered in the log by a single log token, but multiple stream tokens.

3.4 Properties of the vCorfu Stream Store

Materialized streams are a first class abstraction in vCorfu, unlike *streams* in Tango [10] which are merely tags within a shared log. Materialized streams strike a balance that combines the global consistency advantages of shared logs with the locality advantages of dis-

tributed data platforms. Specifically, these properties enable vCorfu to effectively support SMR at scale:

The global log is a single source of scalability, consistency, durability and history. One may wonder, why have log replicas at all, if all we care to read from are materialized streams? First, the global log provides a convenient, scalable mechanism to obtain a consistent snapshot of the entire system. This can be used to execute long running read-only transactions, a key part of many analytics workloads, or a backup utility could constantly scan the log and move it to cold storage. Second, the log provides us with a unique level of fault tolerance - even if all the stream replicas were to fail, vCorfu can fall back to using the log replicas only, continuing to service requests.

Materialized streams are true virtual logs, unlike streams. Tango streams enable clients to selectively consume a set of updates in a shared log. Clients read sequentially from streams using a `readNext()` call, which returns the next entry in the stream. Tango clients cannot randomly read from anywhere in stream because streams are implemented using a technique called *backpointers*: each entry in a stream points to the previous entry, inducing a requirement for sequential traversal. Materializing the stream removes this restriction: since clients have access to a replica which contains all the updates for a given stream, clients can perform all the functions they would call on a log, including a random read given a stream address, or a bulk read of an entire stream. This support is essential if clients randomly read from different streams, as backpointers would require reading each stream from the tail in order.

vCorfu avoids backpointers, which pose performance, concurrency and recovery issues. Backpointers can result in performance degradation when concurrent clients are writing to the log and a timeout occurs, causing a *hole filling protocol* to be invoked [9]. Since holes have no backpointers, timeouts force a linear scan of the log, with a cost proportional to the number of streams in the log. Tango mitigates this problem by keeping the number of streams low and storing multiple backpointers, which has significant overhead because the sequencer must maintain a queue for each stream. Furthermore, backpointers significantly complicate recovery: if the sequencer fails, the entire log must be read to determine the most recent writes to each stream. vCorfu instead relies on stream replicas, which contain a complete copy of updates for each stream, free of holes thanks to vCorfu's commit protocol, resorting to a single backpointer only when stream replicas fail. Sequencer recovery is fast, since stream replicas can be queried for the most recent update.

Stream replicas may handle playback and directly serve requests. In most shared log designs, clients must consume updates, which are distributed and sharded for performance. The log itself cannot directly serve requests because no single storage unit for the log contains all the updates necessary to service a request. Stream replicas in vCorfu, however, contain all the updates for a particular stream, so a stream replica can playback updates locally and directly service requests to clients, a departure from the traditional client-driven shared log paradigm. This removes the burden of playback from clients and avoids the playback bottleneck of previous shared log designs [10, 11].

Garbage collection is greatly simplified. In Tango, clients cannot trim (release entries for garbage collection) streams directly. Instead, they must read the stream to determine which log addresses should be released, and issue trim calls for each log address, which can be a costly operation if many entries are to be released. In vCorfu, clients issue trim commands to stream replicas, which release storage locally and issue trim commands to the global log. Clients may also delegate the task of garbage collection directly to a stream replica.

4 The vCorfu Architecture

vCorfu presents itself as an object store to applications. Developers interact with objects stored in vCorfu and a client library, which we refer to as the vCorfu runtime, provides consistency and durability by manipulating and appending to the vCorfu stream store. Today, the vCorfu runtime supports Java, but we envision supporting many other languages in the future.

The vCorfu runtime is inspired by the Tango [10] runtime, which provides a similar distributed object abstraction in C++. On top of the features provided by Tango, such as *linearizable reads* and transactions, vCorfu leverages Java language features which greatly simplify writing vCorfu objects. Developers may store arbitrary Java objects in vCorfu, we only require that the developer provide a serialization method and to annotate the object to indicate which methods read or mutate the object, as shown in Figure 5.

Like Tango, vCorfu fully supports transactions over objects with stronger semantics than most distributed data stores, thanks to inexpensive global snapshots provided by the log. In addition, vCorfu also supports transactions involving objects not in the runtime's local memory (case D, §4.1 in [10]), *opacity* [22], which ensures that transactions never observe inconsistent state, and *read-own-writes* which greatly simplifies concurrent programming. Unlike Tango, the vCorfu runtime never

```
class User {
    String name; String password;
    DateTime lastLogin; DateTime lastLogout;

    @Accessor
    public String getName() {
        return name;}

    @MutatorAccessor
    public boolean login(String pass, DateTime time){
      if (password.equals(pass)) {
      lastLogin = time;
      return true;}
   return false;}

    @Mutator
    public void logout(DateTime time) {
      lastLogout = time;}}
```

Figure 5: A Java object stored in vCorfu. @Mutator indicates that the method modifies the object, @Accessor indicates the method reads the object, and @MutatorAccessor indicates the object reads and modifies the object.

needs to resolve whether transactional entries in the log have succeeded thanks to a lightweight transaction mechanism provided by the sequencer.

4.1 vCorfu Runtime

To interact with vCorfu as an object store, clients load the vCorfu runtime, a library which manages interactions with the vCorfu stream store. Developers never interact with the store directly, instead, the runtime manipulates the store whenever an object is accessed or modified. The runtime provides each client with a view of objects stored in vCorfu, and these views are synchronized through the vCorfu stream store.

The runtime provides three functions to clients: `open()`, which retrieves a in-memory view of an object stored in the log, `TXbegin()`, which starts a transaction, and `TXend()`, which commits a transaction.

4.2 vCorfu Objects

As we described earlier, vCorfu objects can be arbitrary Java objects such as the one shown in Figure 5. Objects map to a stream, which stores updates to that object.

Like many shared log systems, we use state machine replication (SMR) [27] to provide strongly consistent accesses to objects. When a method annotated with @Mutator or @MutatorAccessor is called, the runtime serializes the method call and appends it to the objects' stream first. When an @Accessor or @MutatorAccessor is called, the runtime reads all the updates to that stream, and applies those updates to the object's state before returning. In order for SMR to work, each mutator must be deterministic (a call to `random()` or `new Date()` is not supported). Many method calls can be easily refactored to take non-deterministic calls as a parameter, as shown in the `login` method in Figure 5.

The SMR technique extracts several important properties from the vCorfu stream store. First, the log acts as a source of *consistency*: every change to an object is totally ordered by the sequencer, and every access to an object reflects all updates which happen before it. Second, the log is a source of *durability*, since every object can be reconstructed simply by playing back all the updates in the log. Finally, the log is a source of *history*, as previous versions of the object can be obtained by limiting playback to the desired position.

Each object can be referred to by the id of the stream it is stored in. Stream ids are 128 bits, and we provide a standardized hash function so that objects can be stored using human-readable strings (i.e., "person-1").

vCorfu clients call `open()` with the stream id and an object type to obtain a view of that object. The client also specifies whether the view should be *local*, which means that the object state is stored in-memory locally, or *remote*, which means that the stream replica will store the state and apply updates remotely (this is enabled by the remote class loading feature of Java). Local views are similar to objects in Tango [10] and especially powerful when the client will read an object frequently throughout the lifespan of a view: if the object has not changed, the runtime only performs a quick `check()` call to verify no other client has modified the object, and if it has, the runtime applies the relevant updates. Remote views, on the other hand, are useful when accesses are infrequent, the state of the object is large, or when there are many remote updates to the object - instead of having to playback and store the state of the object in-memory, the runtime simply delegates to the stream replica, which services the request with the same consistency as a local view. To ensure that it can rapidly service requests, the stream replicas generate periodic checkpoints. Finally, the client can optionally specify a maximum position to open the view to, which enables the client to access the history, version or *snapshot* of an object. Clients may have multiple views of the same object: for example, a client may have a local view of the present state of the object with a remote view of a past version of the object, enabling the client to operate against a snapshot.

4.3 Transactions in vCorfu

Transactions enable developers to issue multiple operations which either succeed or fail atomically. Transactions are a pain point for partitioned data stores since a transaction may span across multiple partitions, requiring locking or schemes such as 2PL [32] or MVCC [35] to achieve consistency.

vCorfu leverages atomic multi-stream appends and global snapshots provided by the log, and exploits the sequencer as a lightweight transaction manager. Transaction execution is optimistic, similar to transactions in shared log systems [10, 11]. However, since our sequencer supports conditional token issuance, we avoid polluting the log with transactional aborts.

To execute a transaction, a client informs the runtime that it wishes to enter a transactional context by calling `TXBegin()`. The client obtains the most recently issued log token once from the sequencer and begins optimistic execution by modifying reads to read from a snapshot at that point. Writes are buffered into a write buffer. When the client ends the transaction by calling `TXEnd()`, the client checks if there are any writes in the write buffer. If there are not, then the client has successfully executed a read-only transaction and ends transactional execution. If there are writes in the write buffer, the client informs the sequencer of the log token it used and the streams which will be affected by the transaction. If the streams have not changed, the sequencer issues log and stream tokens to the client, which commits the transaction by writing the write buffer. Otherwise, the sequencer issues no token and the transaction is aborted by the client without writing an entry into the log. This important optimization ensures only committed entries are written, so that when a client encounters a transactional commit entry, it may treat it as any other update. In other shared log systems [10, 11, 40], each client must determine whether a commit record succeeds or aborts, either by running the transaction locally or looking for a decision record. In vCorfu, we have designed transactional support to be as general as possible and to minimize the amount of work that clients must perform to determine the result of a transaction. We treat each object as an opaque object, since fine-grained conflict resolution (for example, determining if two updates to different keys in a map conflict) would either require the client resolve conflicts or a much more heavyweight sequencer.

Opacity is ensured by always operating against the same global snapshot, leveraging the history provided by the log. Opacity [22] is a stronger guarantee than strict serializability as opacity prevents programmers from observing inconsistent state (e.g. a divide-by-zero error when system invariants prevent such a state from occurring). Since global snapshots are expensive in partitioned systems, these systems [1, 2, 3, 4] typically provide only a weaker guarantee, allowing programs to observe inconsistent state but guaranteeing that such transactions will be aborted. Read-own-writes is another property which vCorfu provides: transactional reads will also apply any writes in the write buffer. Many other systems [1, 4, 10] do not provide this property since it requires writes to be applied to data items. The SMR paradigm, however, enables vCorfu to generate the result of a write in-memory, simplifying transactional programming.

vCorfu fully supports *nested transactions*, where a transaction may begin and end within a transaction. Whenever transaction nesting occurs, vCorfu buffers each transaction's write set and the transaction takes the timestamp of the outermost transaction.

4.4 Querying Objects

vCorfu supports several mechanisms for finding and retrieving objects. First, a developer can use vCorfu like a traditional key-value store just by using the stream id for object as a key. We also support a much richer query model: a set of collections, which resemble the Java collections are provided for programmers to store and access objects in. These collections are objects just like any other vCorfu object, so developers are free to implement their own collection. Developers can take advantage of multiple views on the same collection: for instance a `List` can be viewed as a `Queue` or a `Stack` simultaneously. Some of the collections we provide include a `List`, `Queue`, `Stack`, `Map`, and `RangeMap`.

Collections, however, tend to be very large objects which are highly contended. In the next section, we discuss composable state machine replication, a technique which allows vCorfu to build a collection out of multiple objects.

5 Composable State Machine Replication

In vCorfu, objects may be composed of other objects, a technique which we refer to as composable state machine replication (CSMR). The simplest example of CSMR is a hash map composed of multiple hash maps, but much more sophisticated objects can be created.

Composing SMR objects has several important advantages. First, CSMR divides the state of a single object into several smaller objects, which reduces the amount of state stored at each stream. Second, smaller objects reduce contention and false sharing, providing for higher concurrency. Finally, CSMR resembles how data structures are constructed in memory - this allows us to apply standard data structure principles to vCorfu. For example, a B-tree constructed using CSMR would result in a structure with $O(\log n)$ time complexity for search, insert and delete operations. This opens a plethora of familiar data structures to developers.

Programmers manipulate CSMR objects just as they would any other vCorfu object. A CSMR object starts

```
class CSMRMap<K,V> implements Map<K,V> {
    final int numBuckets;

    int getChildNumber(Object k) {
      int hashCode = lubyRackoff(k.hashCode());
      return Math.abs(hashCode % numBuckets);}

    SMRMap<K,V> getChild(int partition) {
      return open(getStreamID() + partition);}

    V get(K key) {
    return getChild(getChildNumber(key)).get(key);}

    @TransactionalMethod(readOnly = true)
    int size() {
      int total = 0;
      for (int i = 0; i < numBuckets; i++) {
      total += getChild(i).size();}
   return total;}

    @TransactionalMethod
    void clear() {
      for (int i = 0; i < numBuckets; i++) {
      total += getChild(i).clear();}}}
```

Figure 6: A CSMR Java Map in vCorfu. @TransactionalMethod indicates that the method must be executed transactionally.

with a *base object*, which defines the interface that a developer will use to access the object. An example of a CSMR hash map is shown in Figure 6. The base object manipulates *child objects*, which store the actual data. Child objects may reuse standard vCorfu objects, like a hash map, or they may be custom-tailored for the CSMR object, like a B-tree node.

In the example CSMR map shown in Figure 6, the object shown is the base object and the child objects are standard SMR maps (backed by a hash map). The number of buckets is set at creation in the `numBuckets` variable. Two functions, `getChildNumber()` and `getChild()` help the base object locate child objects deterministically. In our CSMR map, we use the Luby-Rakoff [28] algorithm to obtain an improved key distribution over the standard Java `hashCode()` function. Most operations such as `get` and `put` operate as before, and the base object needs to only select the correct child to operate on. However, some operations such as `size()` and `clear()` touch all child objects. These methods are annotated with @TransactionalObject so that under the hood, the vCorfu runtime uses transactions to make sure objects are modified atomically and read from a consistent snapshot. The vCorfu log provides fast access to snapshots of arbitrary objects, and the ability to open remote views, which avoids the cost of playback, enables clients to quickly traverse CSMR objects without reading many updates or storing large local state.

In a more complex CSMR object, such as our CSMR B-tree, the base object and the child object may have completely different interfaces. In the case of the B-tree, the base object presents a map-like interface, while the child objects are nodes which contain either keys or references to other child objects. Unlike a traditional B-tree, every node in the CSMR B-tree is versioned like any other object in vCorfu. CSMR takes advantage of this versioning when storing a reference to a child object: instead of storing a static pointer to particular versions of node, as in a traditional B-tree, references in vCorfu are dynamic. Normally, references point to the latest version of an object, but they may point to any version during a snapshotted read, allowing the client to read a consistent version of even the most sophisticated CSMR objects. With dynamic pointers, all pointers are implicitly updated when an object is updated, avoiding a problem in traditional trees, where an update to a single child node can cause an update cascade requiring all pointers up to the root to be explicitly updated, known as the recursive update problem [42].

6 Evaluation

Our test system consists of sixteen 12 core machines running Linux (v4.4.0-38) with 96GB RAM and 10G NICs on each node with a single switch. The average latency measured by `ping` (56 data bytes) between two hosts is 0.18 ± 0.01 ms when the system is idle. All benchmarks are done in-memory, with persistence disabled. Due to the performance limitations and overheads from Java and serialization, our system was CPU-bound and none of our tests were able to saturate the NIC (the maximum bandwidth we achieved from a single node was 1Gb/s, with 4KB writes).

Our evaluation is driven by the following questions:

- What advantages to we obtain by materializing streams? (§ 6.1)
- Do remote views offer NoSQL-like performance with the global consistency of a shared log? (§ 6.2)
- How does the sequencer act as a lightweight, lock-free transaction manager and offer inexpensive read-only transactions? (§ 6.3)
- How does CSMR keep state machines small, while reducing contention and false conflicts? (§ 6.4)

6.1 vCorfu Stream Store

The design of vCorfu relies on performant materialization. To show that materializing streams is efficient, we implement streams using backpointers in vCorfu with chain replication, similar to the implementation described in Tango [10].

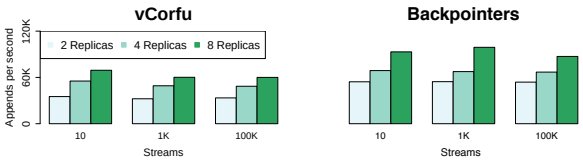


Figure 7: vCorfu's replication protocol imposes a small penalty on writes to support materialization. Each run is denoted with the number of streams used.

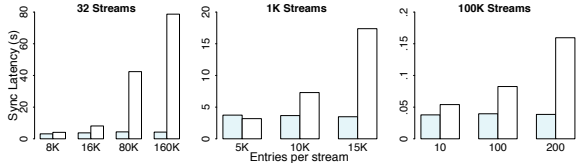


Figure 8: vCorfu enables quick reading of streams. Shaded bars are runs with vCorfu, while white bars represent backpointers. We test with 32, 1K and 100K streams, the number under the bars denotes the number of entries per stream.

For these tests, in order to compare vCorfu with a with a chain replication-based protocol, we use a symmetrical configuration for vCorfu, with an equal number of log replicas and stream replicas. For the backpointer implementation, we use chain replication (i.e. log replicas only), but with the same number of total replicas as the comparison vCorfu system. Our backpointer implementation only stores a single backpointer per entry while Tango uses 4 backpointers. Multiple backpointers are only used to reduce the probability that a linear scan - in tests involving Tango, we disable hole-filling for a fair comparison, except in Figure 9.

The primary drawback of materialization is that it requires writing a commit message, which results in extra messages proportional to the number of streams affected. We characterize the overhead with a microbenchmark that appends 32B entries, varying the number of streams and logging units. Figure 7 shows that writing a commit bit imposes about a 40% penalty on writes, compared to a backpointer based protocol which does not have to send commit messages. However, write throughput continues to scale as we increase the number of replicas, so the bottleneck in both schemes is the rate in which the sequencer can hand out tokens, not the commit protocol.

Now we highlight the power of materializing streams. Figure 8 shows the performance of reading an entire stream with a varying number of 32B entries and streams in the log. The 100K stream case uses significantly fewer entries, reflecting our expectation that CSMR objects will increase the number of streams while decreasing the number of entries per stream. As the number of streams and entries increase, vCorfu greatly outperforms backpointers thanks to the ability to perform a single bulk read, whereas backpointers must traverse the log backwards before being able to serve a single read.

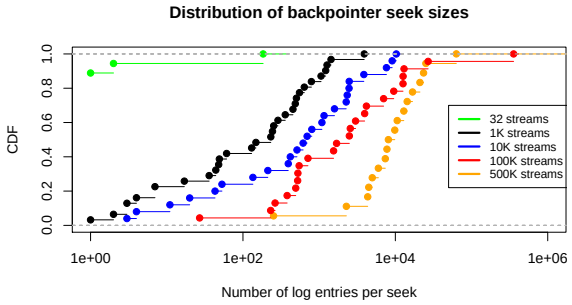


Figure 9: Distribution of the entries that a backpointer implementation must seek through. As the number of streams increases, so does the number of entries that must be scanned as a result of a hole. With 500k streams, there is a 50% chance that 10k entries will have to be scanned due to a hole.

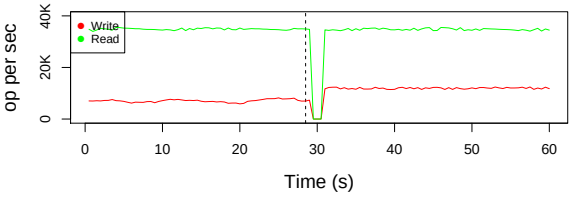


Figure 10: Append and read throughput of a local view during stream replica failure. On the dotted line, we fail a single stream replica. Append throughput increases because the replication factor has now decreased, while read throughput remains constant.

When hole-filling occurs due to client timeouts, backpointers perform very poorly, falling back to a scan because the hole fill does not contain backpointers resulting in a linear scan of the log. Figure 9 examines the number of log entries a backpointer implementation may have to read as a result of a hole. To populate this test, we use 256 clients which randomly select a stream to append a 32B entry to. We then generate a hole, varying the number of streams in the log, and measure the number of entries that the client must seek through. The backpointer implementation cannot do bulk reads, and reading each entry takes about 0.3 ms. The median time to read a stream with a hole takes only 210ms with 32 streams, but jumps to 14.8 and 39.6 seconds with 100K and 500k streams, respectively. vCorfu avoids this issue altogether because stream replicas manage holes.

Finally, Figure 10 shows that vCorfu performance degrades gracefully when a stream replica fails, and vCorfu switches to using the log replicas instead. We instantiate two local views on the same object, and fail the stream replica hosting the object at $t = 29.5s$. The system takes about a millisecond to detect this failure and reconfigure into degraded mode. The append throughput almost doubles, since the replication factor has decreased while the read throughput stays about the same, falling

back to using backpointers. Since the local view contains all the previous updates in the stream, reading the entire stream is not necessary. If a remote view was used, however, vCorfu would have to read the entire stream to restore read access to the object.

6.2 Remote vs. Local Views

Next, we examine the power of remote views. We first show that remote views address the playback bottleneck: In Figure 11, we append to a single local view and increase the number of clients reading from their own local views. As the number of views increases, read throughput decreases because each view must playback the stream and read every update. Once read throughput is saturated readers are unable to keep up with the updates in the system and read latency skyrockets: with just 32 clients, the read latency jumps to above one second. With a remote view, the stream replica takes care of playback and even with 1K clients is able to maintain read throughput and millisecond latency.

We then substantiate our claim that remote views offer performance comparable to many NoSQL data stores. In Figure 12, we run the popular Yahoo! cloud serving benchmark with Cassandra [1] (v 2.1.9), a popular distributed key-value store, as well as the backpointer-based implementation of vCorfu described in the previous section. In vCorfu, we implement a key-value store using the CSMR map described in Section 5 with a bucket size of 1024, and configure the system in a symmetrical configuration with 12 replicas and a chain length of 2. Since the Java map interface returns the previous key (a read-modify-write), we implement a special `fastPut()` method, which does a write without performing a read. For Cassandra, we configure 12 nodes with a replication factor of 2, `SimpleStrategy` replica placement, a consistency level of `ALL` and select the `MurMur3Partitioner` [24]. We turn off synchronous writes to the commit log in Cassandra by setting `durable_writes` to false. The workloads exercised by YCSB are described in Table 2. We configure YCSB with the default 1KB entry size.

vCorfu exhibits comparable write performance to Cassandra - showing that the overhead of the sequencer is low, since both Cassandra and vCorfu must write to two replicas synchronously. However, for reads, Cassandra must read from both replicas in order to not return stale updates, while vCorfu can service the read from the log replica. This leads to significant performance degradation for Cassandra on most of the read-dominated workloads in YCSB. In fact, even with an extra read, Cassandra does not provide the same consistency guarantees as vCorfu as cross-partition reads in Cassandra can still be inconsistent.

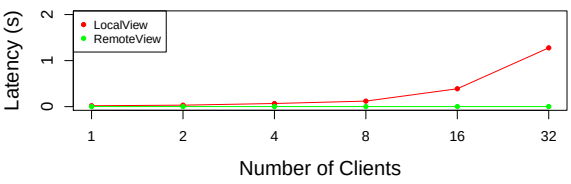


Figure 11: Latency of requests under load with local views and remote views. As the number of clients opening local views on an object increases, so does the latency for a linearized read.

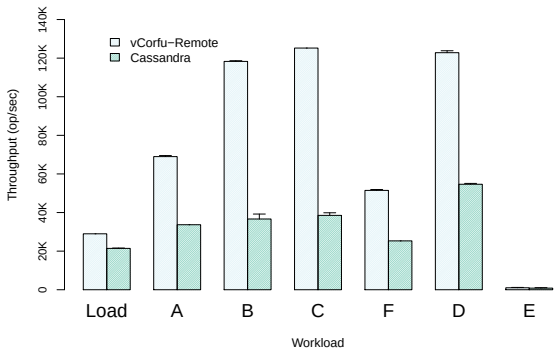


Figure 12: YCSB suite throughput over 3 runs. Error bars indicate standard deviation. Results in order executed by benchmark.

Name	Workload	Description
Load	100% Insert	Propagating Database
A	50% Read/50% Update	Session Store
B	95% Read/5% Update	Photo Tagging
C	100% Read	User Profile Cache
D	95% Read/5% Insert	User Status Updates
E	95% Scan/5% Insert	Threaded Conversations
F	50% Read/50% Read-Modify	User Database

Table 2: YCSB workloads.

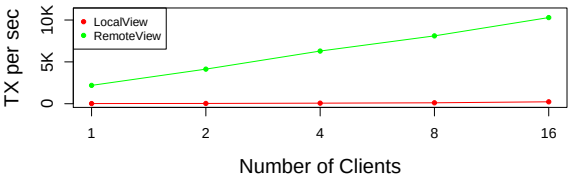


Figure 13: Snapshot transaction performance. The number of snapshot transactions supported scales with the number of client views.

6.3 Transactions

We claimed that vCorfu supports fast efficient transactions through materialization and harnessing the sequencer as a fast, lightweight transaction manager. We demonstrate this by first examining the performance of read-only transactions, and then compare our optimistic and fast-locking transaction design to other transactional systems.

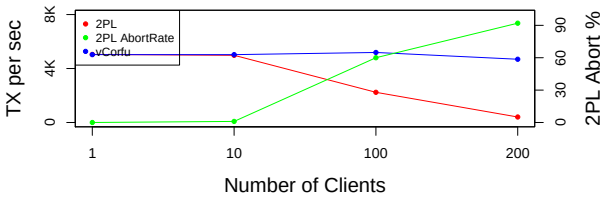


Figure 14: Read-only Transactions Goodput vs. 2PL. As the number of reader clients increase, so does the abort rate for 2PL. In vCorfu, the goodput remains 100%, since read-only transactions never conflict.

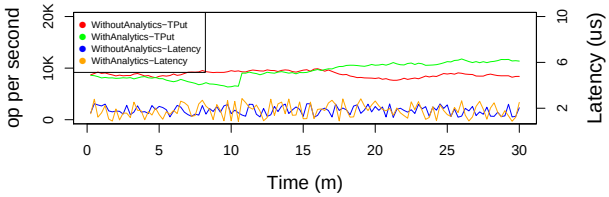


Figure 15: Advertising analytics workload. The system exhibits consistent write performance with a long-running read-only transaction.

We begin our analysis of read-only transactions by running a microbenchmark to show that snapshot transactions scale with the number of clients in vCorfu. In Figure 13, we run snapshot transactions against the YCSB populated database in the previous section with both local views and remote views. Each transaction selects a random snapshot (log offset) to run against, and reads 3 keys. Local views suffer from having to playback the log on each read (they currently do not utilize the snapshots generated by the stream replicas), and can only sustain a few hundred operations per second, while remote views take advantage of the stream replicas ability to perform playback and sustain nearly 10K transactions/s, and scales with the number of clients.

Next, we compare read only transactions to a 2PL approach taken by many NoSQL systems. We use the same Cassandra cluster as in the previous section with a single node ZooKeeper lock service (v 3.4.6) and compare against vCorfu. In the 2PL case, a client acquires locks from ZooKeeper and releases them when the transaction commits. Objects can be locked either for read or for writes, To prevent deadlock, if a transaction cannot acquire all locks, it releases them and aborts the transaction. We use a single writer which selects 10 entries at random to update, and set the target write transaction rate at 5K op/s, and populate each system with 100K objects. We then add an increasing number of reader threads, each which read 10 entries at random. Figure 14 shows that as the number of readers increase, so does the abort rate for the writer thread in the 2PL case, until at 200 concurrent readers, where the abort rate is 92% and the writer thread only can perform a few hundred op/s. In vCorfu, read-only transactions never conflict with other transactions in the system, so the writer throughput remains constant.

We then evaluate vCorfu using a benchmark which models a real-world advertising analytics workload. In this workload, a database tracks the number of views on web pages. Each view contains a small amount of data, including the IP address, and x,y coordinates of each click. Each web page is modeled as an vCorfu object, and the pages views are constantly recorded by a simulator which generates 10K page views/sec. The database tracks a total of 10K pages. We then run a long-running analytics thread, which for 30 minutes, runs a read-only snapshot which iterates over all the web pages, changing the snapshot it is running against every iteration. In Figure 15, we show that running the analytics thread has no impact on write throughput or latency of the system.

6.4 CSMR

Next, we investigate the trade-offs afforded by CSMR. One of the main benefits of CSMR is that it divides large SMR objects into smaller SMR objects, reducing the cost of playback, and reducing the burden on stream replicas. In Figure 16 (top), we compare the performance of initializing a new view and servicing the first read on a in-memory local view of a traditional SMR map, and a CSMR map of varying bucket sizes using 1KB entries. We test using both a uniform key distribution and a skewed Zipf [13] distribution. In a CSMR map, servicing the first read only requires reading from a single bucket, and as the number of buckets increases, the number of updates, and the resulting in-memory state that has to be held is reduced. With 100MB of updates, the SMR map takes nearly 10s to open, while the CSMR maps take no more than 70ms for both the zipf and uniform random distributions, reflecting a 150× speedup.

In addition to keeping the size of the SMR object small, dividing the SMR object through CSMR also reduces contention in the presence of concurrency. Since concurrent writers now touch multiple smaller objects instead of one large object, the chance of conflict is reduced. In Figure 16 (bottom) we compare the abort rate of SMR maps and CSMR maps as before. We perform a transaction which performs two reads and a write over uniformly distributed and zipf distributed keys. Even with only two concurrent writers, transactions on the SMR map must abort half the time. With the CSMR map with 1000 buckets, even with 16 concurrent writers, the abort rate remains at 2%.

Finally, we examine the cost of CSMR: since the state machine is divided into many smaller ones, operations which affect the entire state will touch all the divided objects. To quantify this cost, Figure 17 performs a clear operation, followed by a size operation - which requires

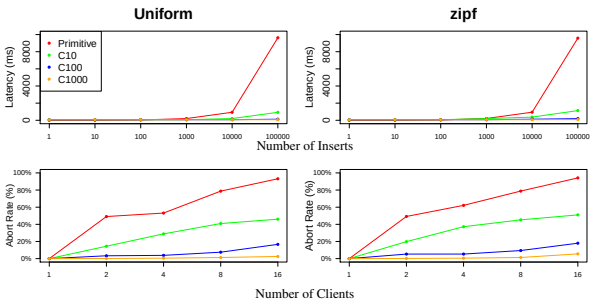


Figure 16: Top: The latency of initializing a local view versus the number of updates to the object, for different bucket sizes and on a primitive SMR map. Bottom: The abort rate of optimistic transactions with varying concurrency and bucket sizes on a primitive SMR map.

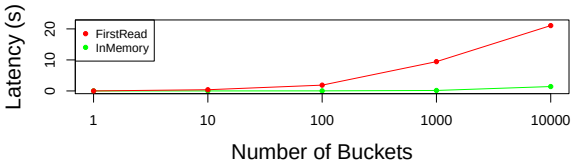


Figure 17: The latency of a clear operation followed by a size operation versus the number of buckets.

a transactional write followed by a transactional read to all buckets. If this operation is performed using remote views, the latency remains relatively low, even with 10K buckets, but can shoot up to almost 30s if a remote view has to playback entries for all buckets. This shows that even with a heavily divided object, remote views keep CSMR efficient even for transactional operations.

7 Related Work

The vCorfu stream store design is inspired by Replex [39] and Hyperdex [20], systems which deviate from traditional replication by placing replicas according to multiple indexes. In vCorfu, we use two indexes: one for the log replicas and the other for a stream replicas. The unique requirement in vCorfu is that these indexes will maintain the same relative ordering for entries: two entries i, j which appear on a stream such that i precedes j must also appear on the global log with i preceding j. This requirement is enforced by the dynamic chain replication protocol described in Section 4.

Whereas vCorfu starts with a global shared log as a source of serialization and virtualizes it, other transactional distributed data platforms do precisely the opposite: They partition data into shards, and build distributed transactions over them. A wide variety of mechanisms were developed for distributed transactions at scale, some providing weaker semantics than serializability [5, 25], others optimize specific kinds of transactions [6]. The cores of serializable transaction systems include variants of two-phase locking [17, 32], a serialization oracle [8, 15], or a two-round distributed ordering protocol [20, 31]. By leveraging both log and stream replicas, vCorfu provides both the benefits of a total order and partitioning at the same time. In particular, vCorfu trivially supports lockless read-only transactions. This turns out to be a desirable capability for long-lived analytics transactions, which has caused considerable added complexity in systems [17, 29].

vCorfu is built around SMR [37], which has been used both with [10, 11] and without [7, 23] shared logs to implement strongly consistent services. The SMR paradigm requires that each replica store a complete copy of the state, which is impractical for replicating large systems at scale, such as databases. vCorfu takes advantage of CSMR to logically partition large state machines, and stream replicas to scale SMR and the shared log to many clients.

vCorfu also shares similarities to log-structured file systems such as LFS [34], btrfs [33] and WAFL [19]. These systems suffer from a recursive update problem [42], which can result in significant write amplification and performance degradation. CSMR avoids this issue, since pointers to vCorfu objects refer to the latest version of the object, no pointer updates are required.

8 Conclusion

Driving transacional data platforms over a shared log is a well understood paradigm in the database world, but has been challenging to scale out in systems like Hyder, Tango, and Calvin; driving data platforms over a scattered collection of logs like Kafka or Kinesis has met serious challenges around consistency and transactions. The vCorfu branch store strikes an ideal balance which marries the global consistency advantages of shared logs with the locality advantages of distributed data platforms. We presented the vCorfu design and implementation and described how it tackles performance challenges in data services with strong and rich atomicity guarantees.

Acknowledgements

This paper is dedicated to all members of the Corfu team, wherever they may be today. Special thanks to our shepherd, Robbert van Renesse for his detailed feedback, as well as our anonymous reviewers, for their helpful feedback and the term *materialized stream*.

References

[1] *Cassandra*. `http://cassandra.apache.org/`.

[2] *Cockroach Labs*. `http://www.cockroachlabs.com/`.

[3] *Couchbase*. `http://www.couchbase.com/`.

[4] *Gemfire*. `http://www.gemfire.com/`.

[5] Marcos K. Aguilera, Joshua B. Leners, and Michael Walfish. Yesquel: Scalable SQL storage for web applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 245–262, Monterey, California, USA, 2015. ACM.

[6] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 27(3):5:1–5:48, November 2009.

[7] Deniz Altinbuken and Emin Gun Sirer. Commodifying replicated state machines with OpenReplica. 2012.

[8] J. Baker, C. Bond, J.C. Corbett, J. Furman, A. Khorlin, J. Larson, J.M. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: providing scalable, highly available storage for interactive services. In *Proceedings of Conference on Innovative Data Systems Research*, CIDR, pages 223–234, Asilomar, CA, USA, 2011.

[9] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. Corfu: A distributed shared log. *ACM Transactions on Computer Systems (TOCS)*, 31(4):10, 2013.

[10] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, Nemacolin, PA, USA.

[11] Philip A Bernstein, Colin W Reid, and Sudipto Das. Hyder-A transactional record manager for shared flash. CIDR, Asilomar, CA.

[12] Josiah L. Carlson. *Redis in Action*. Manning Publications Co., 2013.

[13] William B. Cavnar, John M Trenkle, et al. N-gram-based text categorization. In *3rd Annual Symposium on Document Analysis and Information Retrieval (SDAIR '94)*, volume 48113, pages 161–175, Las Vegas, NV, USA, 1994.

[14] Kristina Chodorow. *MongoDB: the definitive guide*. O'Reilly Media, 2013.

[15] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the Very Large Data Base Endowment*, 1(2):1277–1288, August 2008.

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC)*, pages 143–154, Indianapolis, IN, USA, 2010. ACM.

[17] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Hollywood, CA, USA, 2012. USENIX Association.

[18] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review (OSR)*, 41(6):205–220, 2007.

[19] John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A. Smith, et al. FlexVol: flexible, efficient file volume virtualization in WAFL. In *USENIX 2008 Annual Technical Conference (ATC '08)*, pages 129–142, Boston, MA, USA, 2008. USENIX Association.

[20] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review*, 42(4):25–36, 2012.

[21] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *38th Annual International Symposium on Computer Architecture (ISCA '11)*, pages 365–376, San Jose, CA, USA, 2011. IEEE.

[22] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*, pages 175–184, Salt Lake City, UT, 2008. ACM.

[23] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference (ATC '10)*, Boston, MA, USA.

[24] Markus Klems, David Bermbach, and Rene Weinert. A runtime quality measurement framework for cloud database service systems. In *Quality of Information and Communications Technology (QUATIC), 2012 Eighth International Conference on the*, pages 38–46. IEEE, 2012.

[25] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, Prague, Czech Republic, 2013. ACM.

[26] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[27] Leslie Lamport. The part-time parliament. *FAST*, 3:15–30, 2004.

[28] Michael Luby and Charles Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2):373–386, 1988.

[29] Dahlia Malkhi and Jean-Philippe Martin. Spanner's concurrency control. *ACM SIGACT News*, 44(3):73–77, 2013.

[30] Sajee Mathew. Overview of Amazon Web Services. *Amazon Whitepapers*, 2014.

[31] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI' 14)*, pages 479–494, Broomfield, CO, USA, October 2014. USENIX Association.

[32] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, BC, Canada.

[33] Ohad Rodeh, Josef Bacik, and Chris Mason. btrfs: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[34] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[35] Pierangelo Di Sanzo, Bruno Ciciani, Francesco Quaglia, and Paolo Romano. A performance model of multi-version concurrency control. In *Modeling, Analysis and Simulation of Computers and Telecommunication Systems, 2008. MASCOTS 2008. IEEE International Symposium on*, pages 1–10. IEEE, 2008.

[36] Frank Schmuck and Jim Wylie. Experience with transactions in QuickSilver. In *ACM SIGOPS Operating Systems Review (OSR)*, volume 25, pages 239–253. ACM, 1991.

[37] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[38] Alfred Z. Spector, Joshua J. Bloch, Dean S. Daniels, and Richard P. Draves. The Camelot project. 1986.

[39] Amy Tai, Michael Wei, Michael J. Freedman, Ittai Abraham, and Dahlia Malkhi. Replex: A scalable, highly available multi-index data store. In *2016 USENIX Annual Technical Conference (USENIX ATC '16)*, Denver, CO, June 2016. USENIX Association.

[40] Alexander Thomson and Daniel J. Abadi. CalvinFS: consistent WAN replication and scalable metadata management for distributed file systems. In *13th USENIX Conference on File and Storage Technologies (FAST '15)*, pages 1–14, Santa Clara, CA, 2015.

[41] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, Scottsdale, Arizona, USA, 2012. ACM.

[42] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST '12)*, page 1, Santa Clara, CA, USA, 2012.

Curator: Self-Managing Storage for Enterprise Clusters

Ignacio Cano*[w], Srinivas Aiyar[n], Varun Arora[n], Manosiz Bhattacharyya[n], Akhilesh Chaganti[n], Chern Cheah[n], Brent Chun[n], Karan Gupta[n], Vinayak Khot[n] and Arvind Krishnamurthy[w]

[w]University of Washington
{icano,arvind}@cs.washington.edu
[n]Nutanix Inc.
curator@nutanix.com

Abstract

Modern cluster storage systems perform a variety of background tasks to improve the performance, availability, durability, and cost-efficiency of stored data. For example, cleaners compact fragmented data to generate long sequential runs, tiering services automatically migrate data between solid-state and hard disk drives based on usage, recovery mechanisms replicate data to improve availability and durability in the face of failures, cost saving techniques perform data transformations to reduce the storage costs, and so on.

In this work, we present Curator, a background MapReduce-style execution framework for cluster management tasks, in the context of a distributed storage system used in enterprise clusters. We describe Curator's design and implementation, and evaluate its performance using a handful of relevant metrics. We further report experiences and lessons learned from its five-year construction period, as well as thousands of customer deployments. Finally, we propose a machine learning-based model to identify an efficient execution policy for Curator's management tasks that can adapt to varying workload characteristics.

1 Introduction

Today's cluster storage systems embody significant functionality in order to support the needs of enterprise clusters. For example, they provide automatic replication and recovery to deal with faults, they support scaling, and provide seamless integration of both solid-state and hard disk drives. Further, they support storage workloads suited for virtual machines, through mechanisms such as snapshotting and automatic reclamation of unnecessary data, as well as perform space-saving transformations such as dedupe, compression, erasure coding, etc.

Closer examination of these tasks reveals that much of their functionality can be performed in the background. Based on this observation, we designed and implemented a background self-managing layer for storage in enterprise clusters. As part of this work, we addressed a set of engineering and technical challenges we faced to realize such a system. First, we needed an extensible, flexible, and scalable framework to perform a diverse set of tasks in order to maintain the storage system's health and performance. To that end, we borrowed technologies that are typically used in a different domain, namely big data analytics, and built a general system comprised of two main components: 1) Curator, a background execution framework for cluster management tasks, where all the tasks can be expressed as MapReduce-style operations over the corresponding data, and 2) a replicated and consistent key-value store, where all the important metadata of the storage system is maintained. Second, we needed appropriate synchronization between background and foreground tasks. We addressed these synchronization issues by having the background daemon act as a client to the storage system, and we let the latter handle all the synchronization. Third, minimal interference with foreground tasks was required. We accomplished this by using task priorities and scheduling heuristics that minimize overheads and interference. The resulting framework let us implement a variety of background tasks that enable the storage system to continuously perform consistency checks,[1] and be self-healing and self-managing.

We performed this work in the context of a commercial enterprise cluster product developed by Nutanix.[2] We developed the system over a period of five years, and have deployed it on thousands of enterprise clusters. We report on the performance of the system and experiences gleaned from building and refining it. We found that Curator performs garbage collection and replication effectively, balances disks, and makes storage access efficient through a number of optimizations. Moreover, we realized that the framework was general enough to incorporate a wide variety of background transformations as well as simplified the construction of the storage system.

Nonetheless, we noticed that our heuristics do not necessarily work well in all clusters as there is significant heterogeneity across them. Thus, we recently started developing a framework that uses machine learning (ML) for addressing the issues of when should these background management tasks be performed and how much work they should do. The ML-based approach has two

*This work was done while the author was interning at Nutanix Inc.

[1]This eliminates heavyweight fsck-like operations at recovery time.

[2]Nutanix is a provider of enterprise clusters. For more details refer to http://www.nutanix.com.

key requirements: 1) high predictive accuracy, and 2) the ability to learn or adapt to (changing) workload characteristics. We propose using reinforcement learning, in particular, the Q-learning algorithm. We focus our initial efforts on the following tiering question: how much data to keep in SSDs and HDDs? Empirical evaluation on five simulated workloads confirms the general validity of our approach, and shows up to ∼20% latency improvements.

In summary, our main contributions are:

- We provide an extensive description of the design and implementation of Curator, an advanced distributed cluster background management system, which performs, among others, data migration between storage tiers based on usage, data replication, disk balancing, garbage collection, etc.
- We present measurements on the benefits of Curator using a number of relevant metrics, e.g., latency, I/O operations per second (IOPS), disk usage, etc., in a contained local environment as well as in customer deployments and internal corporate clusters.
- Finally, we propose a model, based on reinforcement learning, to improve Curator's task scheduling. We present empirical results on a storage tiering task that demonstrate the benefits of our solution.

2 Distributed Storage for Enterprise Clusters

We perform our work in the context of a distributed storage system designed by Nutanix for enterprise clusters. In this section, we provide an overview of the software architecture, the key features provided by the storage system, and the data structures used to support them. Herein, we present the necessary background information for understanding the design of Curator.

2.1 Cluster Architecture

The software architecture is designed for enterprise clusters of varying sizes. Nutanix has cluster deployments at a few thousand different customer locations, with cluster sizes typically ranging from a few nodes to a few dozens of nodes. Cluster nodes might have heterogeneous resources, since customers add nodes based on need. The clusters support virtualized execution of (legacy) applications, typically packaged as VMs. The cluster management software provides a management layer for users to create, start, stop, and destroy VMs. Further, this software automatically schedules and migrates VMs taking into account the current cluster membership and the load on each of the individual nodes. These tasks are performed by a *Controller Virtual Machine* (CVM) running on each node in the cluster.

The CVMs work together to form a distributed system that manages all the storage resources in the cluster. The CVMs and the storage resources that they manage provide the abstraction of a distributed storage fabric (DSF) that scales with the number of nodes and provides transparent storage access to user VMs (UVMs) running on any node in the cluster. Figure 1 shows a high-level overview of the cluster architecture.

Applications running in UVMs access the distributed storage fabric using legacy filesystem interfaces (such as NFS, iSCSI, or SMB). Operations on these legacy filesystem interfaces are interposed at the hypervisor layer and redirected to the CVM. The CVM exports one or more block devices that appear as disks to the UVMs. These block devices are virtual (they are implemented by the software running inside the CVMs), and are known as vDisks. Thus, to the UVMs, the CVMs appear to be exporting a storage area network (SAN) that contains disks on which the operations are performed.[3] All user data (including the operating system) in the UVMs resides on these vDisks, and the vDisk operations are eventually mapped to some physical storage device (SSDs or HDDs) located anywhere inside the cluster.

Although the use of CVMs introduces an overhead in terms of resource utilization,[4] it also provides important benefits. First, it allows our storage stack to run on *any* hypervisor. Second, it enables the upgrade of the storage stack software without bringing down nodes. To support this feature, we implemented some simple logic at the hypervisor-level to effectively multi-path its I/O to another CVM in the cluster that is capable of serving the storage request. Third, it provides a clean separation of roles and faster development cycles. Building a complex storage stack in the hypervisor (or even the kernel) would have severely impacted our development speed.

2.2 Storage System and Associated Data Structures

We now describe the key requirements of the DSF and how these requirements influence the data structures used for storing the metadata and the design of Curator.

R1 Reliability/Resiliency: the system should be able to handle failures in a timely manner.

R2 Locality preserving: data should be migrated to the node running the VM that frequently accesses it.

R3 Tiered Storage: data should be tiered across SSDs, hard drives, and the public cloud. Further, the SSD tier should not merely serve as a caching layer for hot data, but also as permanent storage for user data.

R4 Snapshot-able: the system should allow users to quickly create snapshots for greater robustness.

[3] Unlike SAN/NAS and other related solutions (e.g., OneFS [15], zFS [34], GlusterFS [18], LustreFS [39], GPFS [36]), the cluster nodes serve as both VM compute nodes as well as storage nodes.

[4] We are currently exploring some alternatives to reduce such overhead, e.g., pass-through drivers so that the CVMs can handle the disk I/O directly, RDMA to move replication data, etc.

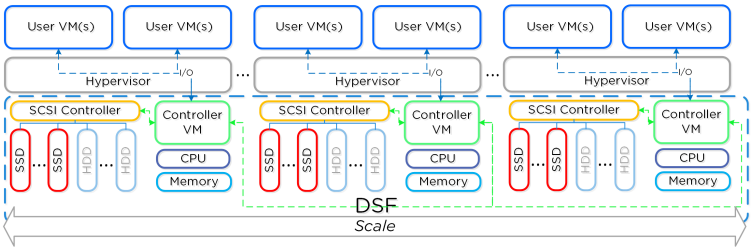


Figure 1: Cluster architecture and the Distributed Storage Fabric. UVMs access the storage distributed across the cluster using CVMs.

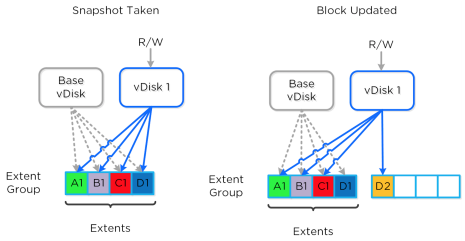


Figure 2: Snapshotting and copy-on-write update to snapshotted extents.

R5 Space efficient: the system should achieve high storage efficiency while supporting legacy applications and without making any assumptions regarding file sizes or other workload patterns.

R6 Scalability: the throughput of the system should scale with the number of nodes in the system.

The above set of requirements manifest in our system design in two ways: (a) the set of data structures that we use for storing the metadata, and (b) the set of management tasks that will be performed by the system. We discuss the data structures below and defer the management tasks performed by Curator to §3.2.

Each vDisk introduced in §2.1 corresponds to a virtual address space forming the individual bytes exposed as a disk to user VMs. Thus, if the vDisk is of size 1 TB, the corresponding address space maintained is 1 TB. This address space is broken up into equal sized units called vDisk blocks. The data in each vDisk block is physically stored on disk in units called extents. Extents are written/read/modified on a sub-extent basis (a.k.a. slice) for granularity and efficiency. The extent size corresponds to the amount of live data inside a vDisk block; if the vDisk block contains unwritten regions, the extent size is smaller than the block size (thus satisfying R5).

Several extents are grouped together into a unit called an extent group. Each extent and extent group is assigned a unique identifier, referred to as extentID and extentGroupID respectively. An extent group is the unit of physical allocation and is stored as a file on disks, with hot extent groups stored in SSDs and cold extent groups on hard drives (R3). Extents and extent groups are dynamically distributed across nodes for fault-tolerance, disk balancing, and performance purposes (R1, R6).

Given the above core constructs (vDisks, extents, and extent groups), we now describe how our system stores the metadata that helps locate the actual content of each vDisk block. The metadata maintained by our system consists of the following three main maps:

- vDiskBlock map: maps a vDisk and an offset (to identify the vDisk block) to an extentID. It is a logical map.
- extentID map: maps an extent to the extent group that it is contained in. This is also a logical map.
- extentGroupID map: maps an extentGroupID to the physical location of the replicas of that extentGroupID and their current state. It is a physical map.

Here are a few implications regarding the aforementioned data structures. Multiple vDisks created through snapshots can share the same extent. The vDiskBlock map of a snapshot can either directly point to an extent shared with a prior snapshot or have a missing entry, in which case the vDiskBlock map of the previous snapshot is consulted. This facility allows for instantaneous creation of snapshots, i.e., we can create an empty vDiskBlock map entry and have it point to the previous snapshot for all of its unfilled entries (R4). At the same time, it enables a later optimization of metadata lookup using lazy filling of the missing entries (§3.2.4). When a vDisk block is updated on the new snapshot, a new extent is created to hold the updated data. Figure 2 shows an example in which vDisk 1 is created as a snapshot and its vDiskBlock map has already been populated with the correct pointers to the corresponding extents (left portion). Later it is updated to point to a new extent upon an update to one of its vDisk blocks (right portion).

The level of indirection introduced by the extentID map allows efficient updates whenever data from one extent group is relocated to another (e.g., to optimize access), as it is a single place in which we store the physical extentGroupID in which the extent is located (thus aiding R2, R3).

Finally, a set of management operations can be performed by only consulting the extentGroupID map. For example, we can detect (and repair) if the number of replicas for a given extentGroupID falls under certain threshold by only accessing this map – the logical maps will remain untouched – thus addressing R1.

Overall, the resulting data structures set us up to perform the various management tasks described in §3.2 in an efficient and responsive manner.

3 Curator

Curator is the cluster management component responsible for managing and distributing various storage management tasks throughout the cluster, including continuous consistency checking, fault recovery, data migration, space reclamation, and many others. In this section, we

describe Curator's architecture (§3.1), the tasks it performs (§3.2), and the policies under which those tasks are executed (§3.3). Finally, we demonstrate its value with a set of empirical results (§3.4), and share our experiences and lessons learned from building Curator (§3.5).

3.1 Curator Architecture

Curator's design is influenced by the following considerations. First, it should scale with the amount of storage served by the storage system and cope with heterogeneity in node resources. Second, Curator should provide a flexible and extensible framework that can support a broad class of background maintenance tasks. Third, Curator's mechanisms should not interfere with nor complicate the operations of the underlying storage fabric. Based on these considerations, we designed a system with the following key components and/or concepts:

Distributed Metadata: The metadata (i.e., the maps discussed in the previous section) is stored in a distributed ring-like manner, based on a heavily modified Apache Cassandra [22], enhanced to provide strong consistency for updates to replicated keys. The decision behind having the metadata distributed lies in the fact that we do not want the system to be bottlenecked by metadata operations. Paxos [23] is utilized to enforce strict consistency in order to guarantee correctness.

Distributed MapReduce Execution Framework: Curator runs as a background process on every node in the cluster using a master/slave architecture. The master is elected using Paxos, and is responsible for task and job delegation. Curator provides a MapReduce-style infrastructure [10] to perform the metadata scans, with the master Curator process managing the execution of MapReduce operations. This ensures that Curator can scale with the amount of cluster storage, adapt to variability in resource availability across cluster nodes, and perform efficient scans/joins on metadata tables.[5]

Although our framework bears resemblance to some data-parallel engines (e.g., Hadoop, Spark), the reason behind writing our own instead of re-purposing an existing one was two-fold: 1) efficiency, as most of these open-source big data engines are not fully optimized to make a single node or a small cluster work efficiently,[6] a must in our case, and 2) their requirement of a distributed storage system (e.g., HDFS), a recursive dependence that we did not want to have in our clustered storage system.

Co-design of Curator with Underlying Storage System: The distributed storage fabric provides an extended API for Curator, including but not limited to the following low-level operations: migrate an extent from one extent group to another, fix an extent group so that it meets the durability and consistency requirements, copy a block map from one vDisk to another, and perform a data transformation on an extent group. Curator only performs operations on metadata, and gives *hints* to an I/O manager service in the storage system to act on the actual data. It is up to the storage system to follow Curator's advice, e.g., it may disregard a suggestion of executing a task due to heavy load or if a concurrent storage system operation has rendered the operation unnecessary.[7] This approach also eliminates the need for Curator to hold locks on metadata in order to synchronize with the foreground tasks; concurrent changes only result in unnecessary operations and does not affect correctness.

Task Execution Modes and Priorities: During a MapReduce-based scan, the mappers and reducers are responsible for scanning the metadata in Cassandra, generating intermediate tables, and creating synchronous and asynchronous tasks to be performed by the DSF. Synchronous tasks are created for fast operations (e.g., delete a vDisk entry in the vDiskBlock metadata map) and are tied to the lifetime of the MapReduce job. Conversely, asynchronous tasks are meant for heavy operations (e.g., dedupe, compression, and replication) and are sent to the master periodically, which batches them, and sends them to the underlying storage system for later execution (with throttling enabled during high load). These tasks are not tied to the lifetime of the MapReduce job. Note that although these tasks are generated based on a cluster-wide global view using MapReduce-based scans, their execution is actually done in the individual nodes paced at a rate suitable to each node's workload.[8] In other words, we compute what tasks need to be performed in a bulk-synchronous manner, but execute them independently (in any order) per node.

3.2 Curator Management Tasks

In this section, we describe how the Curator components work together to perform four main categories of tasks. Table 2 in Appendix A includes a summary of the categories, tasks, and metadata maps touched by each of the tasks.

3.2.1 Recovery Tasks

Disk Failure/Removal (DF) and Fault Tolerance (FT): In the event of a disk or node failure, or if a user simply wants to remove/replace a disk, Curator receives a notification and starts a metadata scan. Such a scan

[5]Note that any metadata stored in a distributed key-value store should be able to utilize this MapReduce framework.

[6]They assume they will have enough compute as their deployments tend to scale out.

[7]Curator makes sure that the I/O manager knows the version of metadata it based its decision on. The I/O manager checks the validity of the operations based on metadata timestamps (for strong consistency tasks like Garbage Collection) or last modified time (for approximate tasks such as Tiering).

[8]The rate depends on the CPU/disk bandwidth available at each node.

finds all the extent groups that have one replica on the failed/removed/replaced node/disk and notifies the underlying storage system to fix these under-replicated extent groups to meet the replication requirement. This is handled by the storage system as a critical task triggered by a high-priority event, which then aims to reduce the time that the cluster has under-replicated data. Note that these tasks require access to just the extentGroupID map and benefit from the factoring of the metadata into separate logical and physical maps.

3.2.2 Data Migration Tasks

Tiering (T): This task moves cold data from a higher storage tier to a lower tier, e.g., from SSD to HDD, or from HDD to the public cloud. Curator is only involved in *down* migration, not *up*, i.e., it does not migrate data from HDD to SSD, or from the public cloud to HDD. *Up* migration, on the other hand, is done by the DSF upon repeated access to hot data. Taken together, the actions of Curator and DSF aim to keep only the hottest data in the fastest storage tiers in order to reduce the overall user access latency.

This task is costly as it involves actual data movement, not just metadata modifications. Curator computes the "coldness" of the data during a metadata scan, and notifies the DSF to perform the actual migration of the coldest pieces. The coldness is computed based on least recently used (LRU) metrics. The cold data is identified by the modified time (mtime) and access time (atime), retrieved during a scan. Both mtime (write) and atime (read) are stored in different metadata maps. The former is located in the extentGroupID map, whereas the latter resides in a special map called extentGroupIDAccess map. This latter access map was especially created to support eventual consistency for non-critical atime data (in contrast to the extentGroupID map's strict consistency requirements) and thereby improve access performance. As a consequence of being stored in separate maps, the mtime and atime of an extent group might be located in different nodes, therefore, communication may be required to combine these two attributes.

In order to compute the "coldness" of the data, a MapReduce job is triggered to scan the aforementioned metadata maps. The *map* tasks emit the extentGroupID as key, and the mtime (or atime) as value. The *reduce* tasks perform a join-like reduce based on the extentGroupID key. The reduce tasks generate the $(egid, mtime, atime)$ tuples for different extent groups and sort these tuples to find the cold extent groups. Finally, the coldest extent groups are sent to the DSF for the actual data migration.

Disk Balancing (DB): Disk Balancing is a task that moves data within the same storage tier, from high usage disks to low usage ones. The goal is to bring the usage of disks within the same tier, e.g., the cluster SSD tier, as close as possible to the mean usage of the tier. This task not only reduces the storage tier imbalance, but also decreases the cost of replication in the case of a node/disk failure. To minimize unnecessary balancing operations, Curator does not execute the balancing if the mean usage is low, even if the disk usage spread is high. Further, in case it executes the balancing, as with Tiering, it only attempts to move cold data. The MapReduce scans identify unbalanced source and target disks, together with cold data, and notifies the storage fabric to perform the actual migration of extent groups.

3.2.3 Space Reclamation Tasks

Garbage Collection (GC): There are many sources of garbage in the storage system, e.g., when an extent is deleted but the extent group still has multiple live extents and cannot be deleted, garbage due to wasting preallocated larger disk spaces on extent groups that became immutable and did not use all of the allocated quota, when the compression factor for an extent group changes, etc. GC increases the usable space by reclaiming garbage and reducing fragmentation. It does so in three ways:

- Migrate Extents: migrate live extents to a new extent group, delete the old extent group, and then reclaim the old extent group's garbage. It is an expensive operation as it involves data reads and writes. Therefore, Curator performs a cost-benefit analysis per extent group and chooses for migration only the extent groups where the benefit (amount of dead space in the extent group) is greater than the cost (sum of space of live extents to be migrated).
- Pack Extents: try to pack as many live extents as possible in a single extent group.
- Truncate Extent Groups: reclaim space by truncating extent groups, i.e., reducing their size.

Data Removal (DR): The data structures introduced in §2.2 are updated in such a way that there cannot be dangling pointers, i.e., there cannot be a vDisk pointing to an extent that does not exist, or an extent pointing to an extent group that does not exist. However, there can be unreachable data, e.g., an extent that is not referenced by any vDisk, or an extent group that is not referenced by any extent. These could be due to the side-effects of vDisk/snapshot delete operations or a consequence of failed DSF operations.

In DSF, extent groups are created first, then extents, and finally vDisks. For removal, the process is backwards; unused vDisks are removed first, then the extents, and finally the unreferenced extent groups. The DR task performs this removal process in stages (possibly in successive scans), and enables the reclamation of unused

space in the system.[9]

3.2.4 Data Transformation Tasks

Compression (C) and Erasure Coding (EC): Curator scans the metadata tables and flags an extent group as a candidate for compression/coding if the current compression of the extent group is different from the desired compression type or if the extent group is sufficiently cold.

Once Curator identifies the extent groups (thus extents) for compression/coding, it sends a request to the DSF, which performs the actual transformation by migrating the extents. The main input parameters of this request are the set of extents to be compressed (or migrated), and the extentGroupID into which these extents will be migrated. If the latter is not specified, then a new extent group is created. This API allows us to pack extents from multiple source extent groups into a single extent group. Also, instead of always creating a new extent group to pack the extents, Curator can select an existing extent group and pack more extents into it. The target extent groups are also identified using MapReduce scans and sorts.

Deduplication (DD): Dedupe is a slightly different data transformation, as it involves accessing other metadata maps. During a scan, Curator detects duplicate data based on the number of copies that have the same precomputed fingerprint, and notifies the DSF to perform the actual deduplication.

Snapshot Tree Reduction (STR): As briefly mentioned in §2.2, the storage system supports snapshots, which are *immutable* lightweight copies of data (similar to a simlink), and can therefore generate an instantaneous copy of a vDisk. Every time the system takes a snapshot, a new node is added to a tree, called the snapshot tree, and the vDisk metadata is inherited. Snapshot trees can become rather deep. In order to be able to read a leaf node from a tree, the system needs to traverse a sequence of vDiskBlock map entries. The bigger the depth of a tree, the more inefficient the read operation becomes.

To address this, the STR task "cuts" the snapshot trees, by copying vDiskBlock map metadata from parents to child nodes. There are two flavors, *partial* and *full* STR, and their use depends on whether we need vDisk metadata only from some ancestors (*partial*) or from all of them (*full*). Once the copy is performed, the child vDisks have all the information needed for direct reads, i.e., there is no need to access the ancestors' metadata, thus, the read latency is reduced.

[9]Note that only the deletion of extent groups frees up physical space.

3.3 Policies

The tasks described in §3.2 are executed based on (roughly) four different policies, described below.

Event-driven: These tasks are triggered by events. For example, whenever a disk/node fails, a Recovery task is executed, no matter what. These are critical, higher priority tasks.

Threshold-based: These are dynamically executed tasks based on fixed thresholds violations. For example, when the tier usage is "high", or the disk usage is "too" unbalanced, etc. We provide both examples below.

In order to be eligible for the Tiering task, the storage tier usage from where we want to down migrate the data should exceed a certain threshold f, whereas the destination tier usage should not exceed a threshold d, i.e., it should have enough space to store the data to be moved. Further, a threshold h indicates by how much the usage percentage is to be reduced.[10]

Regarding DB, in order to be considered for balancing, the mean tier usage should exceed a threshold m and the disk usage spread should be greater than a threshold s. The disk usage spread is the difference between the disk with maximum usage and the disk with minimum usage within the tier.[11]

Periodic Partial: We next consider tasks that are neither triggered nor threshold-driven, but access only a subset of the metadata maps. These tasks are executed every h_1 hours, and are grouped based on the metadata tables they scan.

Periodic Full: All tasks are executed as part of a *full* scan every h_2 hours. We call this policy *full* as it scans all three metadata tables in Cassandra, the vDiskBlock, extentID, and extentGroupID maps. Because the *partial* scan only works on a subset of the metadata maps, it can run more frequently than the *full* scan, i.e., $h_1 < h_2$. In general, scans are expensive, hence, when a scan is running, Curator tries to identify as many asynchronous tasks as possible and lets them drain into the DSF over time. In other words, Curator combines the processing that must be done for the different tasks in order to reduce the scans' overheads.

3.4 Evaluation

In this section, we evaluate Curator's effectiveness with respect to a number of metrics. We report results on three different settings: a) customer clusters, where Curator is always turned on, b) internal corporate production clusters, where Curator is also on, and c) an internal local cluster, where we enable/disable Curator to perform controlled experiments.

[10]The default threshold values are $f = 75\%$, $d = 90\%$, and $h = 15\%$.
[11]The default threshold values are $m = 35\%$ and $s = 15\%$.

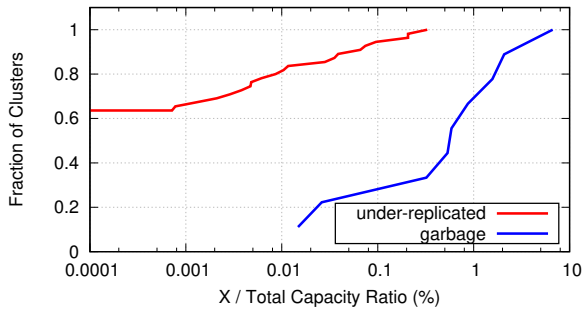


Figure 3: Under-Replicated Data and Garbage percentages (in log-scale) with respect to Total Storage Capacity

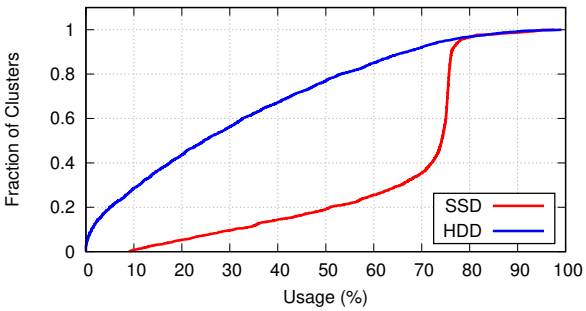


Figure 4: SSD and HDD Usage

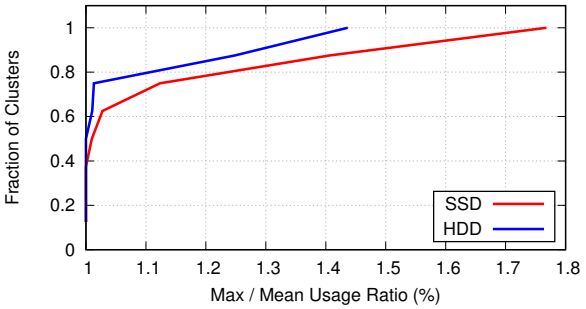


Figure 5: Max/Mean Usage Ratio

3.4.1 Customer and Corporate Clusters

We leverage historical data from a number of real clusters to assess Curator capabilities. In particular, we use ∼50 clusters over a period of two and a half months[12] to demonstrate Curator's contributions to the overall cluster resiliency, and data migration tasks. We also collect data from ten internal corporate clusters over a period of three days. These clusters are very heterogeneous in terms of load and workloads, as they are used by different teams to (stress) test diverse functionalities.

Recovery: Figure 3 shows the cumulative distribution function (CDF) of the average under-replicated data as a percentage of the overall cluster storage capacity (in log-scale) in our customer clusters. We observe that around 60% of the clusters do not present any under-replication problem. Further, 95% of the clusters have at most an average of 0.1% under-replicated data.

For further confirmation, we access the availability cases[13] of the 40% of clusters from Figure 3 that reported under-replication. We consider only those cases for the clusters that were opened within 2 weeks of the under-replication event (as indicated by the metric timestamp), and look for unplanned down time in those clusters. We do not find any unplanned down time in such clusters, which suggests that Curator ensured that replication happened upon detecting the under-replication event so that there was no availability loss.

Tiering: Figure 4 shows the CDF of SSD and HDD usage in our customer clusters. We observe that 40% of the clusters have a SSD usage of at most ∼70-75%. From the remaining 60% of the clusters, many of them have 75% SSD usage, which indicates that the Tiering task is doing its job; data has been down-migrated so that the SSDs can absorb either new writes or up-migration of hot data. In the other 10%, the SSD usage is slightly higher, which means that although the Tiering task is being executed, it cannot entirely cope with such (storage-heavy) workloads. We also note that HDD utilization is typically less, with 80% of clusters having less than 50% HDD usage.

Garbage Collection: Figure 3 also illustrates the CDF of the (95^{th} percentile) percentage of garbage with respect to the total storage capacity (in log-scale) in our corporate clusters. We observe that 90% of the clusters have less than 2% of garbage, which confirms the usefulness of the Garbage Collection task.

Disk Balancing: Figure 5 validates Disk Balancing in our corporate clusters. We plot maximum over mean usage ratio, for both SSDs and HDDs. We observe that in 60% (SSDs) and 80% (HDDs) of the cases, the maximum disk usage is almost the same as the mean.

3.4.2 Internal Cluster

We are interested in evaluating the costs incurred by Curator as well as the benefits it provides, with respect to a "Curator-less" system, i.e., we want to compare the cluster behavior with Curator enabled and when Curator is disabled. Given that we cannot toggle Curator status (ON-OFF) in customer deployments, in this section, we do so in an internal test cluster. We provide a summary of our findings in Table 3 in Appendix B.

Setup: We use a 4-node cluster in our experiments. The cluster has 4 SSDs and 8 HDDs, for a total size of 1.85 TB for SSDs, and 13.80 TB for HDDs, with an overall CPU clock rate of 115.2 GHz, and a total memory of 511.6 GiB.

[12] June to mid August 2016.

[13] We have access to a database of cases information corresponding to various issues encountered in real clusters, where we can query using different filters, e.g., availability problems, etc.

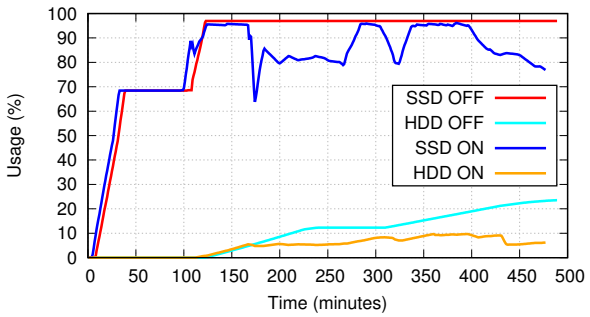


Figure 6: SSD and HDD Usage with Curator ON and OFF

Workloads: We use Flexible I/O Tester (fio[14]) to generate the exact same workloads for testing both settings, i.e., the behavior of the system when Curator is ON and OFF. We re-image the cluster to the same initial clean state when we toggle Curator status.

We simulate three online transaction processing (OLTP) workloads, *small*, *medium*, and *large*, which we execute *sequentially* as part of a single run. Each of these workloads go over three phases, *prefill*, *execution*, and *destroy*. In the *prefill* stage, they create their own user virtual machines (UVMs), together with their associated vDisks. After the *prefill* phase is done, they proceed to *execution*, where the actual workload operations (reads and/or writes) are executed. Following *execution*, the *destroy* stage begins, where the UVMs and associated vDisks are destroyed, i.e., the vDisks' space can be reclaimed. Appendix C describes the workloads in more detail.

Benefits: In terms of benefits, we consider latency and storage usage, which mainly highlight the benefits of T and DR tasks.

Figure 6 shows SSD and HDD usage over time for both Curator ON and OFF. We observe that SSD and HDD usage when Curator is OFF follows a non-decreasing pattern. When SSDs get full (∼125 minutes), all the data starts being ingested directly into HDDs. Instead, when Curator is ON, we see the effects of Tiering, where colder data is moved to HDDs when the default usage threshold is surpassed (§3.3). Even though Tiering kicks in "on time", the data ingestion rate is so high that the task cannot entirely cope with it, therefore, we observe SSD usage percentages in the 90's. At the end, we see that it reaches the 70's.

Figure 6 also illustrates the benefits of Garbage Collection and Data Removal in Curator. When Curator is disabled, we observe a 96% SSD and 23% HDD usage (∼5 TB) at the end of the run, whereas, when Curator is enabled, we see a 76% SSD and 6% HDD usage (∼2.27 TB). The average storage usage over the whole run is ∼2 TB and ∼3 TB for Curator ON and OFF respectively. These differences are mainly due to the DR task (§3.2.3). As described above, the *destroy* phase of each workload, where UVMs and associated vDisks are destroyed, allows the DR task to kick in and start the data removal process, allowing huge storage savings.

Regarding latency, we see an average of ∼12 ms when Curator in ON, and ∼62 ms when is OFF. We measure these values on the *execution* phase of the workloads. As time progresses, the latencies increase when Curator is disabled. We speculate this is due to the fact that the newest ingested data goes directly into HDDs, as SSDs are already full, thus, high latency penalties are paid when reads/writes are issued.

Costs: We consider CPU and memory usage, as well as the number of I/O operations performed.

We see that the number of IOPS executed is higher when Curator is ON, as many of its tasks require reading and writing actual data. Still, the overall average IOPS when Curator is enabled lies in the same ballpark as the disabled counterpart, ∼1400 as opposed to ∼1150 when Curator is OFF.

We also notice that when Curator is ON, the CPU usage is slightly higher. This is due to Curator internals, i.e., its MapReduce infrastructure. Although the mappers primarily scan the metadata (mostly I/O intensive), the reducers involve significant logic to process the scanned information (mostly CPU intensive). Even though the average CPU usage is higher when Curator is enabled, 18% as opposed to 14%, the value is still in an acceptable range, and shows a somewhat stable pattern over time. Regarding memory usage, we do not see a difference between both versions of the system, as shown in Table 3.

3.5 Experiences and Lessons Learned

In this section, we highlight some of the key experiences we gleaned from building Curator.

Firstly, the fact that we had a background processing framework in the form of Curator simplified the addition of new features into the file system. Whenever a new feature was to be added, we systematically identified how that feature could be factored into a foreground component (which would be run as part of the DSF) and a background component (which would be run as part of Curator). This allowed for easy integration of new functionality and kept the foreground work from becoming complex. As an example, our foreground operations do not perform transactional updates to metadata. Instead, they rely on Curator to roll-back incomplete operations as part of its continuous background consistency checks.

Secondly, having a background MapReduce process to do post-process/lazy storage optimization allowed us to achieve better latencies for user I/O. While serving an I/O request, the DSF did not have to make globally optimal

[14] https://github.com/axboe/fio

decisions on where to put a piece of data nor what transformations (compression, dedupe, etc.) to apply on that data. Instead, it could make decisions based on minimal local context, which allowed us to serve user I/O faster. Later on, Curator in the background would re-examine those decisions and make a globally optimal choice for data placement and transformation.

Thirdly, given that we use MapReduce, almost all Curator tasks were required to be expressed using MapReduce constructs (map and reduce operations). For most tasks, this was straightforward and allowed us to build more advanced functionality. MapReduce was however cumbersome in some others, as it required us to scan entire metadata tables during the map phase. We leveraged once again our infrastructure to first filter out which portions of the metadata maps to analyze before performing the actual analysis. This filter step became another map operation, and could be flexibly added to the beginning of a MapReduce pipeline. In retrospect, given our choice of the metadata repository (i.e., a distributed key-value store), we believe MapReduce was the right choice as it provided an easy and efficient way to process our metadata, where we could leverage the compute power of multiple nodes and also ensure that the initial map operations are performed on node-local metadata, with communication incurred only on a much smaller subset of the metadata communicated to the reduce steps.

In terms of the distributed key-value store, although it needed more hard work from the perspective of processing the metadata, it provided us a way to scale from small clusters (say three nodes) to larger (hundreds of nodes) ones. If we had decided to keep the data in a single node, a SQL-Lite like DB could have been enough to do most of the processing we are doing in our MapReduce framework. Many of the other commercial storage products had done this, but we observe two main issues: 1) special dedicated nodes for metadata cause a single point of failure, and 2) the vertical scale up requirement of such nodes – as the physical size of these storage nodes increases with the number of logical entities, they will need to be replaced or upgraded in terms of memory/CPU.[15]

Finally, we noticed a considerable heterogeneity across clusters. While nodes in a cluster are typically homogeneous, different clusters were setup with varying amount of resources. The workload patterns were also different, some ran server workloads, others were used for virtual desktop infrastructure (VDI), whereas some others were deployed for big data applications. Further, the variation in load across different times of day/week was significant in some clusters but not in others. Given this heterogeneity, our heuristics tended to be sub-optimal in many clusters. This motivated us to look at ML approaches to optimize the scheduling of tasks, which we discuss next.

[15] Note that the GFS file-count issue was one of the primary reasons that motivated Colossus [26].

4 Machine Learning-driven Policies

We have described so far an overview of the distributed storage fabric, and delved further into Curator's design and implementation, its tasks and policies of execution, etc. In this section, we propose our modeling strategy, based on machine learning, to improve the threshold-based policies introduced in §3.3. Note that the techniques presented here have not been deployed yet.

We motivate the need for machine learning-driven policies in §4.1. We provide background information on the general reinforcement learning framework we use for our modeling in §4.2.1, and describe with more details Q-learning in §4.2.2. We finally show the results of some experiments on Tiering, our primary use case, in §4.3.

4.1 Motivation

We observed a wide heterogeneity of workloads across our cluster deployments. Given these distinct characteristics of workloads, we noted that the threshold-based execution policies introduced in §3.3 were not optimal for every cluster, nor for individual clusters over time as some of them experienced different workloads at different times (seasonality effects). Thus, in order to efficiently execute Curators management tasks, it became necessary to build "smarter" policies that could adapt on a case-by-case basis at runtime.

The traditional way to improve performance is to use profiling in order to tune certain parameters at the beginning of cluster deployments. Nevertheless, simple profiling would not easily adapt to the varying loads (and changing workloads) our clusters are exposed to over their lifetime. We would need to run profilers every so often, and we would lose, in some sense, past knowledge. We therefore propose using a ML-based solution, which leverages the potential of statistical models to detect patterns and predict future behavior based on the past.

4.2 Background

We encompass this problem within the abstract and flexible reinforcement learning framework, explained in §4.2.1. In particular, we use the model-free popular Q-learning algorithm described in §4.2.2.

4.2.1 Reinforcement Learning (RL)

Reinforcement learning considers the problem of a learning agent interacting with its environment to achieve a goal. Such an agent must be able to sense, to some extent, the state of the environment, and must be able to take actions that will lead to other states. By acting in the world, the agent will receive rewards and/or punishments, and from these it will determine what to do

next [35]. RL is about learning a policy π that maps situations to actions, so as to maximize a numerical reward signal.[16] The agent is not told which actions to take, but instead, it must discover which actions yield the most reward by trying them [42].

More formally, the agent interacts with the environment in a sequence of discrete time steps, $t = 0, 1, 2, 3....$ At each time step t, the agent senses the environment's state, $s_t \in S$, where S is the set of all possible states, and selects an action, $a_t \in A(s_t)$, where $A(s_t)$ is the set of all actions available in state s_t. The agent receives a reward, $r_{t+1} \in \mathbf{R}$, and finds itself in a new state, $s_{t+1} \in S$.

The goal of the agent is to maximize the total reward it receives over the long run. If the sequence of rewards received after time step t is $r_{t+1}, r_{t+2}, r_{t+3}, ...$, then the objective of learning is to maximize the *expected discounted return*. The discounted return G_t is given by:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad (1)$$

where $0 \leq \gamma \leq 1$ is called the discount factor. $\gamma = 0$ will make the agent "myopic" (or short-sighted) by only considering immediate rewards, while $\gamma \to 1$ will make it strive for a long-term high reward [42].

Given that we do not have examples of desired behavior (i.e., training data) but we can assign a measure of goodness (i.e., reward) to examples of behavior (i.e., state-action) [38], RL is a natural fit to our problem.

4.2.2 Q-Learning

Q-Learning [44] is a reinforcement learning algorithm, which falls under the class of temporal difference (TD) methods [40, 41], where an agent tries an action a_t at a particular state s_t, and evaluates its effects in terms of the immediate reward r_{t+1} it receives and its estimate of the value of the state s_{t+1} to which it is taken. By repeatedly trying all actions in all states, it learns which ones are best, i.e., it learns the optimal policy π^*, judged by long-term discounted return.

One of the strengths of this model-free algorithm is its ability to learn without requiring a model of the environment, something model-based approaches do need. Also, model-free methods often work well when the state space is large (our case), as opposed to model-based ones, which tend to work better when the state space is manageable [5].[17]

Q-Learning uses a function Q that accepts a state s_t and action a_t, and outputs the value of that state-action pair, which is the estimate of the expected value (discounted return) of doing action a_t in state s_t and then following the optimal policy π^* [21]. Its simplest form, one-step Q-learning, is given by:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \quad (2)$$

where $0 \leq \alpha \leq 1$ is the learning rate, and determines to what extent new information overrides old one.

Although the learned Q-function can be used to determine an optimal action, the algorithm does not specify what action the agent should actually take [21]. There are two things that are useful for the agent to do, known as the exploration/exploitation trade-off:

- *exploit*: the knowledge that it has of the current state s_t by doing the action in $A(s_t)$ that maximizes $Q(s_t, A(s_t))$.
- *explore*: to build a better estimate of the optimal Q-function. That is, it should select a different action from the one that it currently thinks is the best.

The Q-function above can be implemented using a simple lookup table. Nevertheless, when the state-action space is large, e.g., continuous space, storing Q-values in a table becomes intractable. The Q-function needs to be approximated by a function approximator.

The compression achieved by a function approximator allows the agent to generalize from states it has visited to states it has not. The most important aspect of function approximation is not just related to the space saved, but rather to the fact that it enables generalization over input spaces [35], i.e., the algorithm can now say "the value of these kind of states is x", rather than "the value of this exact specific state is x" [1].

4.3 Use Case: Tiering

Having introduced the basics of reinforcement learning and Q-learning, in this section, we propose using the latter algorithm for deciding when to trigger Tiering. Although our initial efforts are on the tiering question of how much data to keep in SSDs, our approach generalizes to any of the threshold-based tasks described before.

4.3.1 State-Action-Reward

In order to apply Q-learning, we need to define the set of states S, the set of possible actions A, and the rewards r.

We define state s at time step t by the following tuple $s_t = (cpu, mem, ssd, iops, riops, wiops)_t$, where $0 \leq cpu \leq 100$ is the CPU usage, $0 \leq mem \leq 100$ the memory usage, $0 \leq ssd \leq 100$ the SSD usage, $iops \in \mathbf{R}$ the total IOPS, $riops \in \mathbf{R}$ the read IOPS, and $wiops \in \mathbf{R}$ the write IOPS, all at time step t, where $s_t \in S$.

We further define two possible actions for every state, either *not run* or *run* Tiering. Mathematically, the set of actions A is given by $A = \{0, 1\} \forall s_t \in S$, where 0 corresponds to *not run*, and 1 corresponds to *run* the task.

Finally, we use latency as reward. As higher rewards are better, though we prefer lower latencies, we actually

[16] The policy that achieves the highest reward over the long run is known as optimal policy, and typically denoted as π^*.

[17] We were (mainly) inclined to using Q-learning because of these two reasons.

use negative latencies,[18] i.e., the reward r at time step t is given by $r_t = -lat_t$, where $lat_t \in \mathbf{R}$ is the latency in milliseconds at time step t.

4.3.2 Function Approximator

Given that we have a continuous state space S, as defined in §4.3.1, we cannot use a tabular implementation of Q-learning. We therefore resort to a function approximator, and also gain from its advantages towards generalization.

Many approximators have been studied in the past, such as deep neural networks [28, 27], decision trees [33], linear functions [9, 43], kernel-based methods [30], etc. In this work, we choose a linear approximation. The reason behind this decision is two-fold: a) we do not have enough historical data to train more advanced methods, e.g., neural networks (§4.3.3), and b) we see that it works reasonably well in practice (§4.3.4).

4.3.3 Dataset

One key aspect of RL is related to how the agent is deployed. If there is enough time for the agent to learn before it is deployed, e.g., using batch learning with offline historical data, then it might be able to start making right choices sooner, i.e., the policy it follows might be closer to the optimal policy. Whereas if the agent is deployed without any prior knowledge, i.e., has to learn from scratch while being deployed, it may never get to the point where it has learned the optimal policy [21].

We also face this challenge as issues might arise from the time scales associated with online training from scratch in a real cluster; it may take a long time before the state space is (fully) explored. To overcome this limitation, we build a dataset from data collected from a subset of the 50 customer clusters mentioned in §3.4.1. In particular, we use ∼40 clusters, from which we have fine-grained data to represent states, actions, and rewards. The data consists of ∼32K transitions, sampled from the (suboptimal) threshold-based policy. Every cluster was using the same default thresholds described in §3.3. Even using a suboptimal policy to "bootstrap" a model can be helpful in reaching good states sooner, and is a common practice for offline RL evaluation [25].

Following ML practices, we split the dataset into training (80%) and test (20%) sets, and do 3-fold cross validation in the training set for hyper-parameter tuning. We standardize the features by removing the mean and scaling to unit variance. We train two linear models, one for each action, with Stochastic Gradient Descent (SGD) [7] using the squared loss.

4.3.4 Evaluation

In this section, we evaluate our Q-learning model, and compare it to the baseline model, i.e., the threshold-based solution. We use the same internal cluster setup as §3.4.2 to run our experiments.

We deploy our agent "pre-trained" with the dataset described in §4.3.3. Once deployed, the agent keeps on interacting with the environment, exploring/exploiting the state space. We use the popular ε-greedy strategy, i.e., with probability ε the agent selects a random action, and with probability $1-\varepsilon$ the agents selects the greedy action (the action that it currently thinks is the best). We use $\varepsilon = 0.2$ in all our experiments. It is possible to vary ε over time, to favor exploration on early stages, and more exploitation as time progresses. We leave that to future work. Further, we set $\gamma = 0.9$.

Table 1 presents results for five different workloads, described in Appendix D. We compute these numbers based on the *execution* phase of the workloads, i.e., after the pre-fill stage is done, and where the actual read/writes are executed. More results are included in Appendix E. The current experiments are within a short time frame (order of hours), we expect to see even better results with longer runs. We observe that in all of the cases our Q-learning solution reduces the average latency, from ∼2% in the *oltp-varying* workload, up to ∼20% in the *oltp-skewed* one, as well as improves the total number of SSD bytes read. We believe that further improvements could also be achieved by adding more features to the states, e.g., time of the day features to capture temporal dynamics, HDD usage, etc. We also notice that Q-learning demands more IOPS. This is the case since our solution, in general, triggers more tasks than the baseline, thus more I/O operations are performed. Overall, we see that our approach can trade manageable penalties in terms of number of IOPS for a significant improvement in SSD hits, which further translates into significant latency reductions, in most of our experimental settings.

5 Related Work

Our work borrows techniques from prior work on cluster storage and distributed systems, but we compose them in new ways to address the unique characteristics of our cluster setting. Note that our setting corresponds to clusters where cluster nodes are heterogeneous, unmodified (legacy) client applications are packaged as VMs, and cluster nodes can be equipped with fast storage technologies (SSDs, NVMe, etc.). Given this setting, we designed a system where client applications run on the same nodes as the storage fabric, metadata is distributed across the entire system, and faster storage on cluster nodes is effectively used. We now contrast our work with other related work given these differences in execution settings and design concepts.

Systems such as GFS [17] and HDFS [37] are designed for even more scalable settings but are tailored to work with applications that are modified to take advan-

[18] The choice of using negative latencies is rather arbitrary, we could have used their reciprocals instead.

Workload	Metric	Policy	
		fixed threshold	q-learning
oltp	Avg. Latency (ms)	12.48	**10.60**
	SSD Reads (GB)	31.68	**39.16**
	Avg. # of IOPS	**2551.54**	2903.20
oltp-skewed	Avg. Latency (ms)	18.55	**14.91**
	SSD Reads (GB)	151.99	**176.28**
	Avg. # of IOPS	**6686.90**	7221.01
oltp-varying	Avg. Latency (ms)	17.28	**16.95**
	SSD Reads (GB)	469.28	**488.17**
	Avg. # of IOPS	**7884.94**	8192.32
oltp-vdi	Avg. Latency (ms)	15.41	**13.92**
	SSD Reads (GB)	40.83	**41.27**
	Avg. # of IOPS	**4450.18**	5178.13
oltp-dss	Avg. Latency (ms)	61.65	**53.00**
	SSD Reads (GB)	4601.17	**6233.33**
	Avg. # of IOPS	**3105.60**	3239.59

Table 1: Results Summary

tage of their features (e.g., large file support, append-only files, etc.). Further, they do not distribute metadata, since a single node can serve as a directory server given the use of large files and infrequent metadata interactions. These systems do not take advantage of fast storage – all file operations involve network access and the incremental benefits of fast storage on the server side is minimal.

Cluster storage systems such as SAN and NAS also do not co-locate application processes/VMs with servers. They assume a disaggregated model of computing, wherein applications run on client machines and all the data is served from dedicated clusters [15, 34, 18, 39, 36]. These systems provide scalability benefits and a wide variety of features, such as snapshotting [14], which we borrow in our system as well. But the crucial points of differentiation are that our system uses fast local storage effectively through tiering, data migration, and disk balancing. Moreover, we believe that ours is the first system to run a continuous consistency checker which results in significant reductions in downtime.

We use a number of concepts and solutions from distributed systems: MapReduce [10] to perform cluster-wide computations on metadata, Cassandra [22] to store distributed metadata as a key-value store, Paxos [23] to perform leader election for coordination tasks. Interestingly, MapReduce is not a client application running on top of the storage system but rather part of the storage system framework itself.

In recent years, there has been an increasing amount of literature on applying ML techniques to improve scheduling decisions in a wide variety of areas, such as manufacturing [32, 31, 29, 47, 6], sensor systems [20], multicore data structures [12], autonomic computing [45], operating systems [16], computer architecture [19], etc. In Paragon [11], the authors propose a model based on collaborative filtering to greedily schedule applications in a manner that minimizes interference and maximizes server utilization on clusters with heterogeneous hardware. Their work focuses more on online scheduling of end user workloads, whereas ours, concentrates on the background scheduling of cluster maintenance tasks to improve the overall cluster performance.

Wrangler [46] proposes a model based on Support Vector Machines [8] to build a scheduler that can selectively delay the execution of certain tasks. Similar to our work, they train a linear model based on CPU, disk, memory, as well as other system-level features, in an offline-manner, and then deploy it to make better scheduling decisions. In contrast, our offline (supervised) trained model only "bootstraps" the RL one, which keeps on adapting and learning at runtime, i.e., in an online-manner. Smart Locks [13] is a self-tuning spinlock mechanism that uses RL to optimize the order and relative frequency with which different threads get the lock when contending for it. They use a somewhat similar approach, though they target scheduling decisions at a much lower level.

Perhaps the most similar line of work comes from optimal control [24, 2, 3, 4]. The papers by Prashanth et al. [2, 3] propose using RL for tuning fixed thresholds on traffic light control systems. They propose a Q-learning model that adapts to different traffic conditions in order to switch traffic light signals. We use a similar approach but in a different setting, where we learn to better schedule data migration in a multi-tier storage system.

6 Conclusions

Nowadays, cluster storage systems are built-in with a wide range of functionality that allows to maintain/improve the storage system's health and performance. In this work, we presented Curator, a background self-managing layer for storage systems in the context of a distributed storage fabric used in enterprise clusters. We described Curator's design and implementation, its management tasks, and how our choice of distributing the metadata across several nodes in the cluster made Curator's MapReduce infrastructure necessary and efficient. We evaluated the system in a number of relevant metrics, and reported experiences gathered from its five-year period of construction, as well as thousands of deployments in the field. More recently, and given the heterogeneity across our clusters, we focused our attention on building "smarter" task execution policies. We proposed an initial model that uses reinforcement learning to address the issue of when Curator's management tasks should be executed. Our empirical evaluation on simulated workloads showed promising results, achieving up to ∼20% latency improvements.

References

[1] Q-learning with Neural Networks. http://outlace.com/Reinforcement-Learning-Part-3/. Accessed: 2016-09-07.

[2] A., P. L., AND BHATNAGAR, S. Reinforcement Learning With Function Approximation for Traffic Signal Control. *IEEE Trans. Intelligent Transportation Systems 12*, 2 (2011), 412–421.

[3] A., P. L., AND BHATNAGAR, S. Threshold Tuning Using Stochastic Optimization for Graded Signal Control. *IEEE Trans. Vehicular Technology 61*, 9 (2012), 3865–3880.

[4] A., P. L., CHATTERJEE, A., AND BHATNAGAR, S. Adaptive Sleep-Wake Control using Reinforcement Learning in Sensor Networks. In *Sixth International Conference on Communication Systems and Networks, COMSNETS 2014, Bangalore, India, January 6-10, 2014* (2014), pp. 1–8.

[5] ANDREW, M. Reinforcement Learning, Tutorial Slides by Andrew Moore. https://www.autonlab.org/tutorials/rl.html. Accessed: 2017-02-10.

[6] AYTUG, H., BHATTACHARYYA, S., KOCHLET, G. J., AND SNOWDON, J. L. A Review of Machine Learning in Scheduling. *IEEE Transactions on Engineering Management* (1994).

[7] BOTTOU, L. Large-scale Machine Learning with Stochastic Gradient Descent. In *COMPSTAT* (2010).

[8] CORTES, C., AND VAPNIK, V. Support-Vector Networks. *Mach. Learn. 20*, 3 (Sept. 1995), 273–297.

[9] DAYAN, P. The Convergence of TD(λ) for General λ. *Machine Learning 8* (1992), 341–362.

[10] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[11] DELIMITROU, C., AND KOZYRAKIS, C. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), ASPLOS '13, ACM, pp. 77–88.

[12] EASTEP, J., WINGATE, D., AND AGARWAL, A. Smart Data Structures: An Online Machine Learning Approach to Multicore Data Structures. In *Proceedings of the 8th International Conference on Autonomic Computing, ICAC 2011, Karlsruhe, Germany, June 14-18, 2011* (2011), pp. 11–20.

[13] EASTEP, J., WINGATE, D., SANTAMBROGIO, M. D., AND AGARWAL, A. Smartlocks: Lock Acquisition Scheduling for Self-Aware Synchronization. In *Proceedings of the 7th International Conference on Autonomic Computing, ICAC 2010, Washington, DC, USA, June 7-11, 2010* (2010), pp. 215–224.

[14] EDWARDS, J. K., ELLARD, D., EVERHART, C., FAIR, R., HAMILTON, E., KAHN, A., KANEVSKY, A., LENTINI, J., PRAKASH, A., SMITH, K. A., AND ZAYAS, E. FlexVol: Flexible, Efficient File Volume Virtualization in WAFL. In *USENIX 2008 Annual Technical Conference* (2008), ATC'08, USENIX Association, pp. 129–142.

[15] EMC. EMC Isilon OneFS: A Technical Overview, 2016.

[16] FEDOROVA, A., VENGEROV, D., AND DOUCETTE, D. Operating system Scheduling on Heterogeneous Core Systems. In *Proceedings of 2007 Operating System Support for Heterogeneous Multicore Architectures* (2007).

[17] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (2003), SOSP '03, ACM, pp. 29–43.

[18] GLUSTER. Cloud Storage for the Modern Data Center: An Introduction to Gluster Architecture, 2011.

[19] IPEK, E., MUTLU, O., MARTÍNEZ, J. F., AND CARUANA, R. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *Proceedings of the 35th Annual International Symposium on Computer Architecture* (2008), ISCA '08, IEEE Computer Society, pp. 39–50.

[20] KRAUSE, A., RAJAGOPAL, R., GUPTA, A., AND GUESTRIN, C. Simultaneous Placement and Scheduling of Sensors. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks* (2009), IPSN '09, IEEE Computer Society, pp. 181–192.

[21] L., P. D., AND K., M. A. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, 2010.

[22] LAKSHMAN, A., AND MALIK, P. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[23] LAMPORT, L. Paxos Made Simple. In *ACM SIGACT News* (2001), vol. 32, pp. 51–58.

[24] LU, C., STANKOVIC, J. A., SON, S. H., AND TAO, G. Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms. *Journal of Real-Time Systems, Special Issue on Control-Theoretical Approaches to Real-Time Computing 23* (2002), 85–126.

[25] MARIVATE, V. N. *Improved Empirical Methods in Reinforcement Learning Evaluation*. PhD thesis, 2015.

[26] MCKUSICK, M. K., AND QUINLAN, S. GFS: Evolution on Fast-forward. *Queue 7*, 7 (2009), 10:10–10:20.

[27] MNIH, V., KAVUKCUOGLU, K., SILVER, D., GRAVES, A., ANTONOGLOU, I., WIERSTRA, D., AND RIEDMILLER, M. Playing Atari With Deep Reinforcement Learning. In *NIPS Deep Learning Workshop*. 2013.

[28] MNIH, V., KAVUKCUOGLU, K., SILVER, D., RUSU, A. A., VENESS, J., BELLEMARE, M. G., GRAVES, A., RIEDMILLER, M., FIDJELAND, A. K., OSTROVSKI, G., PETERSEN, S., BEATTIE, C., SADIK, A., ANTONOGLOU, I., KING, H., KUMARAN, D., WIERSTRA, D., LEGG, S., AND HASSABIS, D. Human-level Control through Deep Reinforcement Learning. *Nature 518*, 7540 (02 2015), 529–533.

[29] MÖNCH, L., ZIMMERMANN, J., AND OTTO, P. Machine Learning Techniques for Scheduling Jobs with Incompatible Families and Unequal Ready Times on Parallel Batch Machines. *Eng. Appl. Artif. Intell. 19*, 3 (Apr. 2006), 235–245.

[30] ORMONEIT, D., AND SEN, S. Kernel-Based Reinforcement Learning. In *Machine Learning* (1999), pp. 161–178.

[31] PRIORE, P., DE LA FUENTE, D., GOMEZ, A., AND PUENTE, J. A Review of Machine Learning in Dynamic Scheduling of Flexible Manufacturing Systems. *Artif. Intell. Eng. Des. Anal. Manuf. 15*, 3 (June 2001), 251–263.

[32] PRIORE, P., DE LA FUENTE, D., PUENTE, J., AND PARREÑO, J. A Comparison of Machine-learning Algorithms for Dynamic Scheduling of Flexible Manufacturing Systems. *Eng. Appl. Artif. Intell. 19*, 3 (Apr. 2006), 247–255.

[33] PYEATT, L. D., AND HOWE, A. E. Decision Tree Function Approximation in Reinforcement Learning. Tech. rep., Proceedings of the Third International Symposium on Adaptive Systems: Evolutionary Computation and Probabilistic Graphical Models, 1998.

[34] RODEH, O., AND TEPERMAN, A. zFS - A Scalable Distributed File System Using Object Disks. In *IEEE Symposium on Mass Storage Systems* (2003), IEEE Computer Society, pp. 207–218.

[35] RUSSELL, S. J., AND NORVIG, P. *Artificial Intelligence: A Modern Approach*, 2 ed. Pearson Education, 2003.

[36] SCHMUCK, F., AND HASKIN, R. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002), FAST '02, USENIX Association.

[37] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (2010), MSST '10, IEEE Computer Society, pp. 1–10.

[38] SI, J., BARTO, A. G., POWELL, W. B., AND WUNSCH, D. *Handbook of Learning and Approximate Dynamic Programming (IEEE Press Series on Computational Intelligence)*. Wiley-IEEE Press, 2004.

[39] SUN. Lustre File System: High-Performance Storage Architecture and Scalable Cluster File System, 2007.

[40] SUTTON, R. S. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, 1984. AAI8410337.

[41] SUTTON, R. S. Learning to Predict by the Methods of Temporal Differences. In *MACHINE LEARNING* (1988), Kluwer Academic Publishers, pp. 9–44.

[42] SUTTON, R. S., AND BARTO, A. G. *Introduction to Reinforcement Learning*, 1st ed. MIT Press, 1998.

[43] TSITSIKLIS, J. N., AND ROY, B. V. An Analysis of Temporal-Difference Learning with Function Approximation. Tech. rep., IEEE Transactions on Automatic Control, 1997.

[44] WATKINS, C. J. C. H., AND DAYAN, P. Technical Note: Q-Learning. *Mach. Learn. 8*, 3-4 (May 1992), 279–292.

[45] WHITESON, S., AND STONE, P. Adaptive Job Routing and Scheduling. *Eng. Appl. Artif. Intell. 17*, 7 (Oct. 2004), 855–869.

[46] YADWADKAR, N. J., ANANTHANARAYANAN, G., AND KATZ, R. Wrangler: Predictable and Faster Jobs Using Fewer Resources. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), SOCC '14, ACM, pp. 26:1–26:14.

[47] YIH, Y. Learning Real-Time Scheduling Rules from Optimal Policy of Semi-Markov Decision Processes. *International Journal of Computer Integrated Manufacturing* (1992).

A Metadata Maps Accessed by Curator

Category	Task	Metadata Maps		
		vDiskBlock	**extentID**	**extentGroupID**
Recovery	**DF**			x
	FT			x
Data Migration	**T**[19]			x
	DB			x
Space Reclamation	**GC**	x	x	x
	DR	x	x	x
Data Transformation	**C**			x
	EC			x
	DD	x	x	x
	STR	x		

Table 2: Metadata Tables accessed by Curator tasks

B Benefits/Costs Curator ON/OFF

	Metric (Average)	**Curator**	
		OFF	**ON**
Benefits	Latency (ms)	61.73	**12.3**
	Storage Usage (TB)	3.01	**2.16**
Costs	CPU Usage (%)	**14**	18
	Memory Usage (%)	**14.703**	14.705
	# of IOPS	**1173**	1417

Table 3: Benefits/Costs Summary

C OLTP Workloads

Each OLTP workload is composed of two sections, *Data* and *Log*, which emulates the actual data space and log writing separation in traditional Database Management Systems. The *Data* section performs random reads and writes, 50% reads and 50% writes. Further, 10% of its I/O operations, either reads or writes, have 32k block sizes, and 90% 8k blocks. On the other hand, the *Log* section is write only, 10% of the writes are random, and all operations are 32k. The three workloads (*small*, *medium*, *large*) only differ on how much data they read/write, and the number of IOPS, as shown in Table 4.

Workload	**Data**		**Log**	
	Size (GB)	**IOPS**	**Size (GB)**	**IOPS**
small	800	4000	16	200
medium	1120	6000	16	300
large	1120	8000	12	400

Table 4: OLTP Workloads

[19] Also accesses an additional map, as discussed in 3.2.2.

D ML-driven Policies Workloads

We use the following five workloads to test our ML-driven policies:

- *oltp*: same as the OLTP large workload shown in Table 4, with which we intend to simulate a standard database workload.
- *oltp-skewed*: similar to OLTP large, but here the *Data* section is read only, and performs 8k block random reads according to the following distribution: 90% of the accesses go to 10% of the data. It has a working set size of 4480 GB, and performs 32000 I/O operations per second. With this workload we aim to better understand the effects of hot data skewness.
- *oltp-varying*: Alternates between the OLTP medium and small workloads shown in Table 4 every 20 minutes. In this case, we aim to simulate varying loads of the same type of workload within a cluster.
- *oltp-vdi*: Runs the OLTP large workload in one node while the remaining nodes execute VDI-like workloads in 100 VMs each. A VDI-like workload consists of a working set size of 10 GB, split into *Read* and *Write* sections. 80% of the reads in the *Read* section are random, 10% of the read operations have 32k block sizes, and 90% 8k blocks. Regarding the *Write* section, only 20% of the writes are random, and all have 32k block sizes. The writes are done in the last 2 GB of data, whereas the reads range spans the first 8 GB. The total number of IOPS per VM is 26 (13 each section). Here, the idea is to simulate (concurrent) heterogeneous workloads within a cluster.
- *oltp-dss*: Alternates between the OLTP medium and a DSS (Decision Support System) workload every 20 minutes. As DSS is a type of DB workload, it also has *Data* and *Log* sections, but as opposed to an OLTP workload, the *Data* section of a DSS-like workload performs only reads. Here, such section has a working set size of 448 GB, 100% of the reads are sequential, the read operations size is 1 MB, and the total number of IOPS is 2880. As regards to the *Log* section, the working set is 16 GB, 10% of the writes are random, the block sizes are 32k, and the rate of IOPS is 800. In this case, we attempt to simulate that the workload itself changes over time.

E ML-driven Policies Results

Impact of Tiering on Total Number of SSD Reads

Figure 7 shows the evolution of SSD reads for the *oltp* and *oltp-skewed* workloads. We observe that our method based on Q-learning performs on average more SSD reads (∼8 GB and ∼24 GB respectively) than the baseline for both workloads, which is a consequence of performing more Tiering operations during periods when the offered load from applications is low.

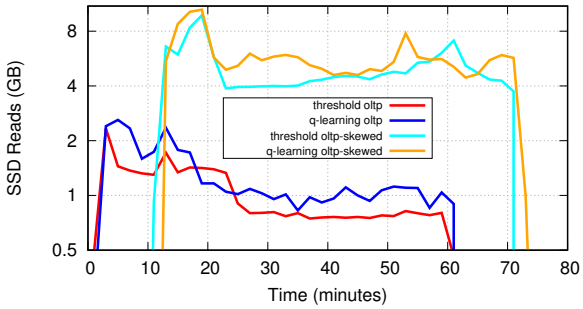


Figure 7: SSD Reads in the *oltp* and *oltp-skewed* workloads

Q-Learning in Action

We now provide performance data corresponding to a sample scenario and illustrate how the Q-Learning model operates in practice. Figure 8 shows the IOPS, latency, and scheduling decisions made while executing the *oltp-dss* workload with our ML-based scheduler. We only plot the *execution* phase of the workload, where the actual operations are executed. Our system polls the state of the cluster every 30 seconds, if it had not triggered Tiering recently, in order to assess whether Tiering should be performed. After a Tiering task is triggered, we wait for 5 minutes before making a new decision, as we do not want to schedule Tiering tasks back-to-back. Regarding the scheduling plot, we not only include the two choices the algorithm makes, *run* and *not run*, but also differentiate whether its decision was due to exploitation (solid lines) or exploration (dashed lines).

The workload keeps on alternating between the OLTP medium and DSS workloads every 20 minutes, as described in Appendix D. It starts with the former, continues with the latter, and so on. We observe that the OLTP medium workload, in general, demands more IOPS than the DSS one, and also achieves lower latencies (cyclic behavior).

At the beginning, even with high IOPS and low latency, the algorithm thinks that the best option is to trigger Tiering (0-20 minutes). When the DSS workload commences (early 20s), the algorithm still keeps scheduling Tiering tasks. In this case, it makes more sense as the cluster utilization is not too high but the latency is. Around minute 44, the algorithm explores the state space by not running Tiering (dashed red line that almost overlaps the solid red ones that follow). Given that this exploration seems to have found a "nice state" with low latency, it considers the best option is to not to run Tiering (first chunk of solid red lines around minute 45). Note that given our 30 seconds polling interval when we do not run Tiering, these lines seem to overlap.

At approximately the 47th minute, the algorithm per-

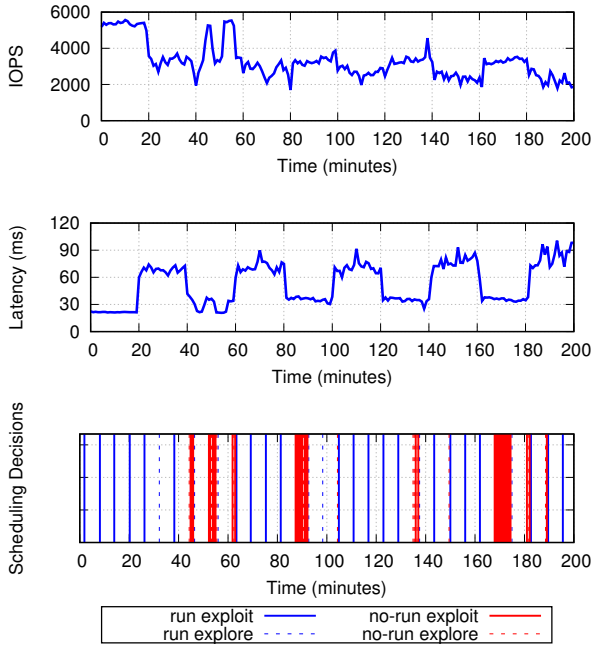


Figure 8: IOPS/Latency/Scheduling Decisions in the *oltp-dss* workload using Q-learning

forms an exploration that triggers Tiering (dashed blue line). It does not work out, as later on, the best decisions are still not to run Tiering (solid red lines around minutes 52-54). Around minute 63, when DSS commences again, the algorithm thinks it is best to run Tiering. At this point, the cluster is not very utilized, i.e., low IOPS, but the latency is high.

The key thing to notice is that the algorithm seems to be learning that when the cluster is highly utilized (high IOPS) and the latency is low, it should not trigger Tiering. During the first period (0-20mins), it was not aware of that, thus it ran Tiering, but later on, it started to figure it out (e.g., 40-60mins and 80-100mins periods). Even more noticeable is between the period 160-180mins, where we observe *many* solid red lines (which appears as a single thick one due to the 30 seconds interval). The 120-140mins period is somewhat surprising. We would have expected more solid red lines there, but they only start appearing towards the end of the period. We believe the algorithm makes early mistakes (minutes 123 and 128), and given that we wait for 5 minutes after running Tiering, it can only realize later on ($\sim$133), where it decides that it is actually better not to run.

Acknowledgments

We are grateful to our reviewers and colleagues for their help and comments on earlier versions of this paper. Special thanks to our shepherd Mosharaf Chowdhury for his valuable feedback. We are also grateful to Hinal Gala, Peihong Huang, and Jason Chan from the Curator QA team for providing very useful information for testing the system. Further, thanks to Chris Wilson from the Performance team for his help in generating the workloads. This work was supported by the National Science Foundation (CNS-1318396, CNS-1420703, CNS-1616774, and CNS-1614717). One of the authors was supported by the Argentine Ministry of Science, Technology and Productive Innovation with the program BEC.AR.

Evaluating the Power of Flexible Packet Processing for Network Resource Allocation

Naveen Kr. Sharma[*] *Antoine Kaufmann*[*] *Thomas Anderson*[*] *Changhoon Kim*[†]
Arvind Krishnamurthy[*] *Jacob Nelson*[‡] *Simon Peter*[§]

Abstract

Recent hardware switch architectures make it feasible to perform flexible packet processing inside the network. This allows operators to configure switches to parse and process custom packet headers using flexible match+action tables in order to exercise control over how packets are processed and routed. However, flexible switches have limited state, support limited types of operations, and limit per-packet computation in order to be able to operate at line rate.

Our work addresses these limitations by providing a set of general building blocks that mask these limitations using approximation techniques and thereby enabling the implementation of realistic network protocols. In particular, we use these building blocks to tackle the network resource allocation problem within datacenters and realize approximate variants of congestion control and load balancing protocols, such as XCP, RCP, and CONGA, that require explicit support from the network. Our evaluations show that these approximations are accurate and that they do not exceed the hardware resource limits associated with these flexible switches. We demonstrate their feasibility by implementing RCP with the production Cavium CNX880xx switch. This implementation provides significantly faster and lower-variance flow completion times compared with TCP.

1 Introduction

Innovation in switch design now leads to not just faster but *more flexible* packet processing architectures. Whereas early generations of software-defined networking switches could specify forwarding paths on a per-flow basis, today's latest switches support configurable per-packet processing, including customizable packet headers and the ability to maintain state inside the switch. Examples include Intel FlexPipe [26], Texas Instruments's Reconfigurable Match Tables (RMTs) [13], the Cavium XPliant switches [15], and Barefoot's Tofino switches [10]. We term these switches *FlexSwitches*.

FlexSwitches give greater control over the network by exposing previously proprietary switching features. They can be reprogrammed to recognize, modify, and add new header fields, choose actions based on user-defined match rules that examine arbitrary components of the packet header, perform simple computations on values in packet headers, and maintain mutable state that preserves the results of computations across packets. Importantly, these advanced data-plane processing features operate at line rate on every packet, addressing a major limitation of earlier solutions such as OpenFlow [22] which could only operate on a small fraction of packets, e.g., for flow setup. FlexSwitches thus hold the promise of ushering in the new paradigm of a *software defined dataplane* that can provide datacenter applications with greater control over the network's datapaths.

Despite their promising new functionality, FlexSwitches are not all-powerful. Per-packet processing capabilities are limited and so is stateful memory. There has not yet been a focused study on how the new hardware features can be used in practice. Consequently, there is limited understanding of how to take advantage of FlexSwitches, nor is there insight into how effective are the hardware features. In other words, do FlexSwitches have *instruction sets* that are sufficiently powerful to support the realization of networking protocols that require in-network processing?

This paper takes a first step towards addressing this question by studying the use of FlexSwitches in the context of a classic problem: network resource allocation. We focus on resource allocation because it has a rich literature that advocates *per-packet dataplane processing* in the network. Researchers have used dataplane processing to provide rate adjustments to end-hosts (congestion control), determine meaningful paths through the network (load balancing), schedule or drop packets (QoS, fairness), monitor flows to detect anomalies and resource exhaustion attacks (IDS), and so on. Our goal is to evaluate the feasibility of implementing these protocols on a concrete and realizable hardware model for programmable data planes.

We begin our study with a functional analysis of the set of issues that arise in realizing protocols such as the RCP [17] congestion control protocol on a FlexSwitch. At first sight, the packet processing capabilities of FlexSwitches appear to be insufficient for RCP and associated protocols; today's FlexSwitches provide a limited number of data plane operators and also limit the number of operations performed per packet and the amount of state maintained by the switch. While some of these limits can be addressed through hardware redesigns that support an enhanced set of operators, the limits associated with switch state and operation count are likely harder

[*]University of Washington
[†]Barefoot Networks
[‡]Microsoft Research
[§]University of Texas at Austin

to overcome if the switches are to operate at line rate. We therefore develop a set of building blocks designed to mask the absence of complex operators and to overcome the switch resource limits through the use of approximation algorithms adapted from the literature on streaming algorithms. We then demonstrate that it is possible to implement approximate versions of classic resource allocation protocols using two design principles. First, our approximations only need to provide an appropriate level of accuracy, taking advantage of *known workload properties* of datacenter networks. Second, we *co-design* the network computation with end-host processing to reduce the computational demand on the FlexSwitch.

Our work makes the following contributions:

- We identify and implement a set of building blocks that mask the constraints associated with FlexSwitches. We quantify the tradeoff between resource use and accuracy under various realistic workloads.
- We show how to implement a variety of network resource allocation protocols using our building blocks.
- In many instances, our FlexSwitch realization is an approximate variant of the original protocol, so we evaluate the extent to which we are able to match the performance of the unmodified protocol. Using ns-3 simulations, we show that our approximate variants emulate their precise counterparts with high accuracy.
- Using an implementation on a production FlexSwitch and emulation on an additional production FlexSwitch hardware model, we show that our protocol implementations are feasible on today's hardware.

2 Resource Allocation Case Study

We begin by studying whether FlexSwitches are capable of supporting RCP, a classic congestion control protocol that assumes switch dataplane operations. We first describe an abstract FlexSwitch hardware model based on existing and upcoming switches. We then examine whether we can implement RCP on the FlexSwitch model. We identify a number of stumbling blocks that would prevent a direct implementation.

FlexSwitch hardware model. Rather than supporting arbitrary per-packet computation, most FlexSwitches provide a *match+action* (M+A) processing model: match on arbitrary packet header fields and apply simple packet processing actions [13]. To keep up with packets arriving at line rate, switches need to operate under tight real-time constraints. Programmers configure how FlexSwitches process and forward packets using high-level languages such as P4 [12].

We assume an abstract switch model as described in [12] and depicted in Figure 1. On packet arrival, FlexSwitches parse packet headers via a user-defined parse graph. Relevant header fields along with packet meta-

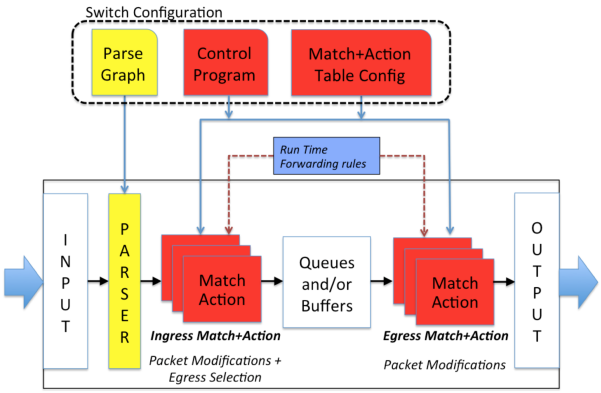


Figure 1: Abstract Switch Model (from [12]).

data are passed to an ingress pipeline of user-defined M+A tables. Each table matches on a subset of extracted fields and can apply simple processing primitives to any field, ordered by a user-defined control program. The ingress pipeline also chooses an output queue for the packet, determining its forwarding destination. Packets pass through an egress pipeline for destination-dependent modifications before output. Legacy forwarding rules (e.g., IP longest prefix match) may also impact packet forwarding and modification.

To support packet processing on the data path, FlexSwitches provide several hardware features:

Computation primitives that perform a limited amount of processing on header fields. This includes operations such as addition, bit-shifts, hashing, and max/min.

M+A tables generalize the abstraction of match+action provided by OpenFlow, allowing matches and actions on any user-defined packet field. Actions may add, modify, and remove fields.

A limited amount of *stateful memory* can maintain state across packets, such as counters, meters, and registers, that can be used while processing.

Switch meta-data, such as queue lengths, congestion status, and bytes transferred, augment stateful memory and can also be used in processing.

Timers built into the hardware can invoke the switch-local CPU to perform periodic computation such as updating M+A table entries or exporting switch meta-data off-switch to a central controller.

Queues with strict priority levels can be assigned packets based on packet header contents and local state. Common hardware implementations support multiple queues per egress port with scheduling algorithms that are fixed in hardware. However, the priority level of any packet can be modified in the processing pipeline, and packets can be placed into any of the queues.

Packets may be sent to a *switch-local control plane CPU* for further processing, albeit at a greater cost. The CPU is able to perform arbitrary computations, for example on packets that do not match in the pipeline. It

cannot guarantee that forwarded packets are processed at line rate; instead, it drops packets when overloaded.

RCP. Rate Control Protocol (RCP) [17] is a feedback-based congestion control algorithm that relies on explicit network feedback to reduce flow completion times. RCP attempts to emulate processor sharing on every switch and assigns a single maximum throughput rate to all flows traversing each switch. In an ideal scenario, the rate given out at time t is simply $R(t) = C/N(t)$, where C is the link capacity and $N(t)$ is the number of ongoing flows, and the rate of each flow is the minimum across the path. Each switch computes $R(t)$ every *control interval*, which is typically set to be the average round-trip time (RTT) of active flows.

In RCP, every packet header carries a rate field R_p, initialized by the sender to its desired sending rate. Every switch on the path to the destination updates the rate field to its own $R(t)$ if $R(t) < R_p$. The receiver returns R_p to the source, which throttles its sending rate accordingly.

The packet header also carries the source's current estimate of the flow RTT. This is used by each switch to compute the control interval. For a precise calculation, per-flow RTTs need to be kept[1].

The original RCP algorithm computes the fair rate $R(t)$ using the equation

$$R(t) = R(t-d) + \frac{\alpha \cdot S - \beta \cdot \frac{Q}{d}}{\hat{N}(t)}$$

where d is the control interval, S is the spare bandwidth, Q is the persistent queue size and α, β are stability constants. $\hat{N}(t)$ is the estimated number of ongoing flows. This *congestion controller* maximizes the link utilization while minimizing queue depth (and drop rates). If there is spare capacity available (i.e., $S > 0$), RCP increases the traffic allocation. If queueing is persistent, RCP decreases rates until queues drain.

FlexSwitch constraints. Although FlexSwitches are flexible and reconfigurable in several ways, they do impose several restrictions and constraints. If these limits are exceeded, then the packet processing code cannot be compiled to the FlexSwitch target. We describe these limits, providing typical values for them based on both published data [13] and information obtained from manufacturers, and note how they impact the implementation of network resource allocation algorithms, like RCP.

Processing primitives are limited. Each switch pipeline stage can execute only one ALU instruction per packet field, and the instructions are limited to signed addition and bitwise logic. Multiplication, division, and floating point operations are not feasible. Hashing primitives, however, are available; this exposes the hardware that switches now use for ECMP and related load-balancing protocols. Control flow mechanisms, such as loops and pointers, are also unavailable, and entries inside M+A tables cannot be updated on the data path. This precludes the complex floating-point computations often employed by resource allocation algorithms from being used directly on the data path. It also limits the number of sequential processing steps to the number of M+A pipeline stages (generally around 10 to 20). Within a pipeline stage, rules are processed in parallel.

Available stateful memory is constrained. Generally, it is infeasible to maintain per-flow state across packets in both reconfigurable switches and their fixed function counterparts.[2] For example, common switches support SRAM-based exact match tables of up to 12 Mb per pipeline stage. The number of rules is limited by the size of the TCAM-based ternary match tables provided per pipeline stage (typically up to 1Mb). In contrast, it is common for a datacenter switch to handle tens of thousands to hundreds of thousands of connections. This allows for a negligible amount of per-connection state, which is likely not enough for most resource allocation algorithms to perform customized flow processing (e.g., for RCP to compute a precise average RTT).

State carried across computational stages is limited. The hardware often imposes a limit on the size of the temporary *packet header vector*, used to communicate data across pipeline stages. Common implementations limit this to 512 bytes. Also, switch metadata may not be available at all processing stages. For example, cut-through switches may not have the packet length available when performing ingress processing. This precludes that information from being used on the data path, severely crimping link utilization metering and flow set size estimation as needed by RCP.

FlexSwitch evolution. It is likely that the precise restrictions given in this section will be improved with future versions of FlexSwitch hardware. For example, additional computational primitives might be introduced or available stateful memory might increase. However, restrictions on the use of per-flow state, the amount of processing that can be done, and the amount of state that can be accessed per packet will have to remain to ensure that the switch can function at line rate.

3 A Library of Building Blocks

To recap, realizing RCP requires additional operations that are not directly supported by FlexSwitches. Specifically, an implementation of RCP requires the following basic features:

[1] However, it is possible to approximate the average RTT and only keep an aggregate of the number of flows.

[2] The dramatic growth in link bandwidths coupled with a much slower growth in switch state resources implies that this constraint will likely hold in future generations of datacenter switches.

- *Metering utilization and queueing:* For every outgoing port on a FlexSwitch, given a pre-specified control interval, we need a metering mechanism that provides the average utilization and the average queueing delay.
- *Estimating the number of flows:* For every outgoing port and a pre-specified control interval, we need an estimate of the number of flows that were transmitted through the port in the previous control interval—without keeping per-flow state.
- *Division/multiplication:* They are required to compute $R(t)$, but are not supported in today's FlexSwitches.

We performed a similar functional analysis of a broad class of resource allocation algorithms to identify their computational and storage requirements. Table 6 presents our findings. We can see that the requirements of most network resource allocation protocols can be distilled into a relatively small set of common *building blocks* that can enable the implementation of these algorithms on a FlexSwitch.

We outline the design of these and other related building blocks in this section. The building blocks are designed to overcome hardware limitations described in Section 2. In particular, we facilitate the following types of packet processing functionality: (a) flow-level measurements such as approximate maintenance of per-flow state and approximate aggregation of statistics across flows, (b) approximate multiplication and division using simpler FlexSwitch primitives, and (c) switch-level measurements such as metering of port-level statistics and approximate scans over port-level statistics.

A common theme across many of these building blocks is the use of approximation to stay within the limited hardware resource bounds. We apply techniques adapted from the streaming algorithms literature within the context of FlexSwitches. Streaming algorithms use limited state to approximate digests as data is streamed through. Often, there is a tradeoff between the accuracy of the measurement and the amount of hardware resources devoted to implementing the measurement. We draw attention to these tradeoffs and evaluate them empirically. We do so by measuring the resource use of faithful C/C++ implementations of the building blocks under various configuration parameters. In later sections, we use P4 specifications of these building blocks, evaluate performance using a production FlexSwitch, and measure resource requirements in an emulator for an additional production FlexSwitch.

3.1 Flow Statistics

Approximating Aggregate Flow Statistics: Many resource allocation algorithms require the computation of aggregate values such as the total number of active flows, the total number of sources and destinations communicating through a switch, and so on. Exact computation of these values would require large amounts of both state and per-packet computation. We therefore design a **cardinality estimator** building block that can approximately calculate the number of unique elements (typically tuples of values associated with a subset of header fields) in a stream of packets in a given time frame.

Our approach is adapted from streaming algorithms [28]. We calculate a hash of each element and count the number of leading zeros n in the binary representation of the result using a TCAM. Using this approach, we compute and store the maximum number $\max(n)$ of leading zeros over all elements to be counted. $2^{\max(n)}$ is then the expected number of unique elements. To implement this calculation on FlexSwitches, we use a ternary match table described by the string:

$$0^n 1 x^{N-n-1} \qquad 0 \leq n < N$$

where N is the maximum number of bits in the hash result. A stateful memory cell maintains the maximum number of leading zeros observed in the stream. This approach allows us to both update and retrieve the count n on the data plane.

This basic approach is space-efficient but exhibits high variance for the estimate. The variance can be reduced by using multiple independent hashes to derive independent estimates of the cardinality. These independent estimates are averaged to produce a final estimate. An alternative is to divide the incoming stream into k disjoint buckets. For example, we can use the last $log(k)$ bits of the hash result as a bucket index. We can estimate the cardinality of each bucket separately and then combine them to derive the final estimate by taking the harmonic mean [28].

Different traffic properties can be estimated by providing different header fields to the hash function. For example, if the header fields are the 4-tuple of source IP, destination IP, source port, and destination port, then we can estimate the number of active flows traversing a switch. Similarly, we can calculate the number of unique end-hosts/VMs/tenants traversing the switch or collect traffic statistics associated with anomalies.

The parameters are the number h of hash functions to use, the maximum number N of bits in each hash result, and the number k of hash buckets to use. Given these parameters, we can ask: how much switch state is required to achieve a given level of accuracy and how does accuracy trade off with switch state?

Prior datacenter studies [4, 31], measure the number of concurrent flows per host to be 100-1000. This means a ToR switch can have 1,000-50,000 flows. Assuming we use a single hash function with $N = 32$, and 1 byte of memory for counting the number of leading zeroes in each bucket, Figure 2 shows the tradeoff of accuracy versus memory usage for different flow set sizes. We vary

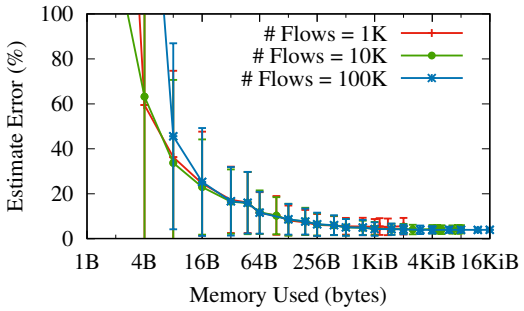


Figure 2: Average tradeoff between accuracy and memory used by the cardinality estimating building block for different set sizes of random flows over 1000 measurements (error bars show standard deviation).

memory use, which equals $h \times k$ bytes, for $k \in [1, 2^{16}]$ and $h \in [1, 8]$ and show the tradeoff for the best choice of h and k for a particular memory size.

Accuracy improves with increased memory, but the flow set size does not impact this tradeoff significantly. Using less than 64 bytes produces error rates above 20%. Memory between 64 bytes and 1 KB has average error rates between 5% and 15%. Using more than 1 KB of memory only improves average error rates marginally (to 4%), but it can improve worst-case error rates.

We do lose accuracy for very small ($\#Flows \approx k$) and very large ($\#Flows \approx 2^N$) flow set sizes. For very small flow sets, several buckets will remain empty. This introduces distortions which can be corrected by calculating the estimate as $\log(k/\#emptyBuckets)$ instead.

In summary, the cardinality estimation building block can work for a wide range of packet property sets with low error and moderate resources, while providing a configurable tradeoff between accuracy and resource use.

Approximating Per-Flow Statistics: Resource limits prevent maintaining accurate per-flow state, so this class of building blocks approximates flow statistics. We discuss two building blocks. The first building block provides per-flow counters. It can be used, for example, to track the number of bytes transmitted by individual flows to identify elephant flows. The second building block tracks timestamp values for a particular flow, e.g. to detect when a flow was last seen for flowlet detection.

Per-flow Counters: For this building block we use a count-min sketch [16] to track element counts in sub-linear space. It supports two operations: `update(e,n)` adds `n` to the counter for element `e` and `read(e)` reads the counter for element `e`. This building block requires a table of r rows and c columns, and r independent hash functions. `update(e,n)` applies the r hash functions to `e` to locate a separate cell in each row, and then adds `n` to each of those r cells. `read(e)` in turn uses the r hash functions to find the same r cells for `e` and returns the minimum. The approximation of the count returned by `read` is always greater than or equal to the exact count. To keep the sketch from saturating, we use the switch CPU to periodically multiply all values in the counter table by a factor < 1 (i.e., decay the counts across multiple intervals), or reset the whole table to 0, depending on the use-case.

N	False-positive	r	c	$r \times c$
1,000	10%	4	512	2,048
	5%	3	1,024	3,072
	1%	3	2,048	6,144
10,000	10%	2	8,192	16,384
	5%	3	8,192	24,576
	1%	4	16,384	65,536
100,000	10%	3	65,536	196,608
	5%	2	131,072	262,144
	1%	4	131,072	524,288

Table 1: Required count-min sketch size to achieve false-positive rate below the specified threshold. The workload consists of N flows that send a total of 10 M packets, 80% of which belong to 20% of the flows—the heavy hitters (HH)— and the HH-threshold is set based on the expected number of packets sent from a HH in this scenario.

Per-Flow Timestamps: To track the timestamp of the last received packet along a flow, we slightly modify the use of the count-min sketch to derive a *min-timestamp* sketch. Instead of adding `n` (the current time) to the selected cells, `update(e,n)` now just overwrites these cells with `n`. `read` remains unmodified and as such returns a conservative estimate for the last time an element has been seen.

One use-case for this block is flowlet switching, wherein packets of a live flowlet should all be routed along the same route but new flowlets can be assigned a new route. Flowlet switching requires two pieces of information per flow: the timestamp of the previous packet, and the assigned route. Our timestamp building block tracks flow timestamps, but storing the route information requires an extension of the building block.

The extended building block supports the following two operations: `update(e,n,r)` sets the timestamp of element `e` to `n` and, if the flowlet timestamp had expired, sets the route to `r`; and `read(e)` returns both the last timestamp and the route for element `e`. We assume that the chosen route for a flow can be represented as a small integer that represents its index in a deterministic set of candidate routes for the flow. Both `update` and `read` still use the same mechanism for finding r cells for a particular `e`, but we extend those cells to store both the timestamp and a route integer. `read` returns the minimum of the timestamps and *the sum* of the route integers from the cells. `update` still updates the timestamps in all selected cells, but will only modify the route information in cells where the timestamp is older than the con-

figured flowlet timeout. If no cell has a timestamp older than the timeout, then the flowlet is considered live and thus the route is not modified. Any non-live flowlet will have at least one cell that has a timestamp older than the timeout, thus the route values stored in these cells can be modified so that the sum of all route values equals the specified `r`. (The pseudocode for this building block is shown in Figure 3.1 in the Appendix.) The key observation for correctness is that the route value in a cell with a live timestamp will never be modified, which guarantees that none of the route information for a live flowlet is modified, because the flowlet's cells will all have live timestamps.

3.2 Approximating Arithmetic

We design a class of building blocks that provide an exact implementation of complex arithmetic where possible and approximations for the general case. We can exactly compute multiplication and division with a constant operand using bit shifts and addition. We resort to approximation only when operands cannot be represented as a short sequence of additions. Our approximate arithmetic supports two variable operands and relies on lookup tables and addition.

Bit shifts: Multiplication and division by powers of 2 can be implemented as bit shifts. If the constant operand is not a power of 2, adding or subtracting the results from multiple bit shifts provides the correct result, e.g., $A \times 6$ can be rewritten as $A \times 2 + A \times 4$. Division can be implemented by rewriting as multiplication with the inverse. Resource constraints (such as the number of pipeline stages) limit the number of shifts and additions that can be executed for a given packet.

Logarithm lookup tables: Where multiplication and division cannot be carried out exactly (e.g., if both operand are variables), we use the fact that $A \times B = \exp(\log A + \log B)$. Given log and exp, we can reduce multiplication and division to addition and subtraction. Because FlexSwitches do not provide log and exp natively, we approximate them using lookup tables.

We use a ternary match table to implement logarithms. For N-bit numbers and a window of calculation accuracy of size m, with $1 \leq m \leq N$, table entries match all bitstrings of the form:

$$0^n 1 (0|1)^{min(m-1,N-n-1)} x^{max(0,N-n-m)} \qquad 0 \leq n < N$$

where x is the wildcard symbol. These entries map to the l-bit log value—represented as a fixed point integer—of the average number covered by the match. For example, for $N = 3$ and $m = 1$, the entries are $\{001, 01x, 1xx\}$, and for $m = 2$ the table entries are $\{001, 010, 011, 10x, 11x\}$. `exp` is calculated using an exact match table that maps logarithms back to the corresponding integer value. The parameters m and l control the space/accuracy tradeoff for this building block. For N-bit operands, table size is approximately $N \times 2^m$ for the log table and precisely 2^l for the exp table. Table 2 shows the minimal values of m and l to achieve a mean relative error below the specified thresholds. Note that even for 64-bit numbers a mean relative error below 1% can be achieved within 2048 exact match and ternary match table entries.

N	Error	m	l	exp (SRAM)	log (TCAM)
16	10%	3	6	64	59
	5%	4	7	128	111
	1%	6	9	512	383
32	10%	3	7	128	123
	5%	4	8	256	239
	1%	6	10	1024	895
64	10%	3	9	512	251
	5%	4	9	512	495
	1%	6	11	2048	1919

Table 2: Required number of lookup table entries for an approximation of multiplication/division for different size operands (N in bits) for the smallest configuration (m and l) with a mean relative error below the specified threshold.

3.3 Switch Statistics

Metering queue lengths: Network protocols often require information about queue lengths. FlexSwitches provide queue length information as part of the packet metadata, but only in the egress pipeline and only for the queue just traversed. This building block tracks queue lengths and makes them available through the entire processing pipeline. When a packet arrives in the egress pipeline we record the length of queue traversed in a register for that queue. Depending on the use-case, this building block can be configured to track the minimal queue length seen in a specified time interval, and using a timer to reset the register at the end of every interval. A variant of this building block is to calculate a continuous exponentially weighted moving average (EWMA) of queue lengths, and this utilizes the approximate arithmetic building block to perform the weighting.

Metering Rates: Similarly, data rates are an important metric in resource allocation schemes. FlexSwitches provide metering functionality in hardware, but the output is generally limited to colors that are assigned based on thresholds as described in the P4 language. This building block is able to measure rates for arbitrary input events in a more general fashion. We provide two configurations: One measures within time intervals, the other measures continuously. We describe the latter as the former is straightforward.

In order to support continuous measurement of the rate during the last time interval T, we use two registers to store the rate and the timestamp of the last update. When updating the estimate, we first calculate the time passed

since the last update as t_Δ. Because there were no events during the last t_Δ ticks, we can calculate how many events occurred during the last time interval T based on the previous rate R as $Y = R \times (T - t_\Delta)$ (or 0 if $t_\Delta > T$). Based on that and the number of events x to be added with the current operation we update the rate to $\frac{Y+x}{T}$, and also update the timestamp register with the current timestamp. The multiplication with R for calculating Y is implemented using the second method described in the approximate multiplication building block, while the division by T can be implemented using a right shift.

Approximate Port Balancing: A common task is to assign packets to one of multiple destinations (e.g., links or servers) to balance load between them. We provide a class of building blocks that implement different balancing mechanisms. The building blocks in this class fall in two separate categories, static balancing steers obliviously to the current load while dynamic balancing takes load into account. We describe the latter.

Making a load balancing decision while taking into account the current load avoids the imbalances common with static load balancing schemes. Computational limitations on FlexSwitches make it infeasible to pick the least loaded destination from a set of candidates larger than 2-4. Previous work [8] has shown that picking the least loaded destination from a small subset – even just 2 – of destinations chosen at random, significantly reduces load imbalances. Picking random candidates can be implemented on FlexSwitches by calculating two or more hashes on high-entropy inputs (timestamp, or other metadata fields). Information about the load of different destinations can be obtained from the *metering queue lengths* or *metering rates* building blocks.

3.4 Discussion

We note that our building blocks address both short and long-term limitations associated with FlexSwitches. For example, some of our building blocks emulate complex arithmetic using simpler primitives and help make the case for supporting some of these complex operations in future versions of FlexSwitches. The rest of the building blocks address fundamental constraints associated with switch state and operation count, thereby allowing the realization of a broader class of protocols that are robust to approximations (as we will see in the next section).

We provide the building blocks in a *template* form, parameterized by zero or more packet header fields. Each block either rewrites the packet header or maintains state variables that are available to subsequent building blocks in the pipeline. For example, the cardinality estimator can be parameterized by the 5-tuple or just the source address and exposes a single variable called 'cardinality'. This variable can be re-used when blocks are chained within a pipeline.

Tables 5 and 6 in Appendix A summarize our building blocks and the different classes of protocols we are able to support with them. We can realize a variety of schemes ranging from classical congestion control and scheduling to load balancing, QoS, and fairness. This shows our blocks are general enough to be reused across multiple protocols and sufficient to implement a broad class of resource allocation protocols.

4 Realizing Network Resource Allocation Protocols

In this section we describe and evaluate a number of network resource allocation protocols that can be built using the building blocks described in Section 3. The network protocols that we target fall into the following broad categories: congestion control, load balancing, QoS/Fairness, and IDS/Monitoring.

4.1 Evaluation Infrastructure

To show that we achieve our goal of implementing complex network resource allocation problems on limited hardware resources using only approximating building blocks, we first implement RCP on top of a real FlexSwitch. We use the Cavium Xpliant CNX880xx [14], a fully flexible switch that provides several of the features described in Section 2 while processing up to 3.2 Tb/s. We then evaluate the accuracy of our implementations versus the non-approximating originals using ns-3 simulations. Finally, we evaluate the resource usage of our implementations and discuss whether they fit within the limited resources that are expected of FlexSwitches.

To use the CNX880xx, we realize various building blocks on the Cavium hardware. This involved: 1) programming the parser to parse protocols, 2) creating tables for stateful memory, 3) configuring the pipeline to perform packet processing operations, and 4) coding the service CPU to perform periodic book-keeping.

To measure the performance of our RCP implementation against other protocols, we emulate a 2-level Fat-Tree topology consisting of 8 servers, 4 ToRs and 2 core switches by creating appropriate VLANs on the switch and directly interconnect the corresponding switch ports. This way, the same switch emulates all switches in the topology. All links operate at 10Gbps. We generate flows from all servers towards a designated target server with Poisson arrivals such that the ToR links are 50-60% utilized. The flows are Pareto distributed ($\alpha = 1.2$), with a mean size of 25 packets. We measure the flow completion times and compare it with the default Linux TCP-cubic implementation.

To evaluate the accuracy of our use cases, we implement them within the ns-3 network simulator [25] version 3.23. We simulate a datacenter network topology consisting of 2560 servers and a total of 112 switches.

The switches are configured in a 3-level FatTree topology with a total over-subscription ratio of 1:4. The core and aggregation switches each have 16×10Gbps ports while the ToRs each have 4×10Gbps ports and 40×1Gbps ports. All servers follow an on-off traffic pattern, sending every new flow to a random server in the datacenter at a rate such that the core link utilization is approximately 30%. Flows are generated using a Pareto distribution ($\alpha = 1.2$), and a mean flow size of 25 packets. Simulations are run for long enough that flow properties can be evaluated.

To evaluate the resource use of our use cases on an actual FlexSwitch, we implement our use cases in the P4 programming language and compile them to a production switch target. The compiler implements the functionality proposed in [19] and compiles to the hardware model described in Section 2. It reports the hardware resource usage for the entire use case, including memory used for data and code.

Our compiler-based evaluation serves to quantify the increased resource usage of our congestion control implementations when added to a baseline switch implementation based upon [34] that provides common functionality of today's datacenter switches. The baseline switch implementation provides basic L2 switching (flooding, learning, and STP), basic L3 routing (IPv4, IPv6, and VRF), link aggregation groups (LAGs), ECMP routing, VXLAN, NVGRE, Geneve and GRE tunneling, and basic statistics collection. We intentionally do not add more functionality to the baseline to highlight the additional resources consumed by our implementations.

4.2 Simple use-cases

The simple use-cases typically apply building blocks in a direct way to achieve their goals. We provide here a few examples to give insight into how our building blocks apply to a wide variety of different network applications.

WCMP. Weighted Cost Multipath (WCMP) routing [39] is an improvement over ECMP that can balance traffic even when underlying network performance is not symmetric. WCMP uses *weights* to express path preferences that consequently impact load balance. To implement WCMP using FlexSwitches, we use the approximate port balancing building block over a next-hop hash table and replicate next-hop entries according to their weight. The WCMP weight reduction algorithm [39] applies in the same way to reduce table entries.

CoDel. This QoS mechanism [24] monitors the minimum observed per-packet queueing delay during a time interval and drops the very last packet in the interval if this minimum observed delay is above a threshold. If there is a packet drop, the scheme provides a formula for calculating a shorter interval for the next time period. This scheme can be easily implemented on a FlexSwitch using metering, a small amount of state for maintaining the time period, approximate arithmetic for computing the next interval, and timers.

TCP port scan detection. To detect TCP port scans, we filter packets for set SYN flags and use the cardinality estimation building block to estimate the number of distinct port numbers observed. If this estimate exceeds a set threshold, we report a scan event to the control plane.

NTP amplification attack detection. Similarly, to detect NTP amplification attacks, we detect NTP packets (UDP port 123) and use cardinality estimation of distinct source IP addresses. If the number is high, we conclude that a large number of senders is emitting NTP requests over a small window of time and report an attack event.

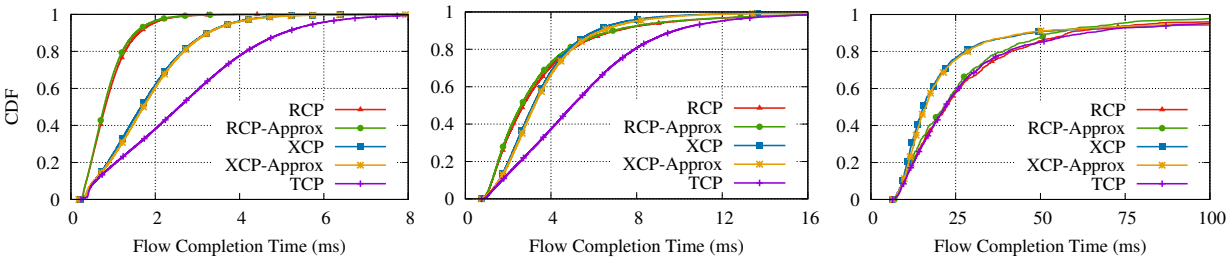
4.3 RCP

We now illustrate how to orchestrate several of the building blocks described in Section 3 to implement RCP. Staying true to our design principles, we employ approximation when necessary to make the implementation possible within limited resource bounds. We recall: To implement RCP we require a way of metering average utilization and queueing delay over a specified control interval. We also need to determine the number of active flows for each outgoing port over the same interval.

First, we estimate the spare capacity and the persistent queues built up inside the switch using the *metering utilization and queueing* building block. We use the building block to maintain per-link meta-data for the number of bytes received and the minimum queue length observed during a control period. When a packet arrives, we update Q by taking the minimum of the previous Q value and the current queue occupancy size as measured by our building block. Similarly, a counter B accumulates the number of bytes sent over the link. A timer is initialized to d, the expected steady state RTT of most flows. When the timer expires, we calculate the spare bandwidth as $S = C - B/d$, where C is the total capacity of the link. d is rounded down to a power of two so that the division operation can be replaced by a bit-shift operation. This use of a slightly smaller d is consistent with RCP's recommendation of using control periods that are roughly comparable to the average RTT.

RCP approximates the number of flows using a circular definition: $\hat{N}(t) = \frac{C}{R(t-d)}$. This essentially estimates the number of flows using the rate computed in the previous control interval. We use a more direct approximation of the number of unique flows by employing the *cardinality estimator* building block.

Finally, with estimates of utilization, queueing, and number of flows, we can use the RCP formula for calculating $R(t)$. Given that RCP is stable for a wide range of $\alpha, \beta > 0$, we pick fractional values that can be approximated by bit shift operations. For the division by $N(t)$, a



(a) Flows of size < 50 packets (b) Flows of size 50-500 packets (c) Flows of size > 500 packets

Figure 3: Cumulative distribution of FCTs of various flow sizes for TCP as well as both precise and approximate RCP and XCP.

general division is required because both sides are variables. We use the *approximate divider* building block to perform the division with sufficient accuracy.

Our hardware implementation of RCP uses several features of the XPliant CNX880xx switch. First, we use the configurable counters to build the cardinality estimator. An array of counters indexed by the packet hash is incremented on each packet arrival. The counter values are read periodically to estimate the number of ongoing flows. Next, we use the metering block to maintain port level statistics such as bytes transmitted and queue lengths. For periodically computing RCP rates, we utilize an on-switch service CPU to compute and store the rates inside the pipeline lookup memory. Finally, we program the reconfigurable pipeline to correctly identify and parse an RCP packet, extract the rate and rewrite it with the switch rate if it is lower.

We compared the performance of the above RCP implementation against TCP on the Cavium testbed and workload described in Section 4.1. We measure the flow completion times for various flow sizes and report them in Table 3. As expected, RCP benefits significantly from the switch notifying the endhosts the precise rate to transmit at. This avoids the need for slow-start and keeps the queue occupancy low at the switches, leading to lower flow completion times.

In order to measure the impact of approximation on the accuracy of our FlexSwitch implementation of RCP, we compare our implementation (RCP-Approx) to an original implementation (RCP) using the simulated workload described in Section 4.1. Figure 3 shows the result as a number of CDFs of flow completion time (FCT) for flows of various lengths. We can see that RCP-Approx matches the performance of RCP closely, for all three types of traffic flows. We also compare to the performance of TCP to validate that our simulation adequately models the performance of RCP. We see that this is indeed the case as RCP performance exceeds TCP performance for shorter flows, as shown in [17].

To measure the additional hardware resources used for RCP-Approx, we integrate RCP-Approx into our baseline switch implementation and compile using the compiler described in Section 4.1. Table 4 shows the additional hardware resources used compared to the baseline switch. We can see that additional resource use is small—not exceeding 6% for all resources but pipeline stages and requiring an additional stage.

Flow Size	TCP Mean	TCP 50th%	TCP 95th%	RCP Mean	RCP 50th%	RCP 95th%
Short	10.32	0.85	2.92	0.85	0.73	2.05
Medium	67.97	5.33	216.99	5.07	3.18	14.85
Long	649.25	59.73	3559.85	50.42	36.85	137.26

Table 3: Flow completion times for short, medium, and long flows (< 50, < 500, and ≥ 500 packets) in milli-seconds on a two-tier topology running RCP on Cavium hardware.

Resource	Baseline	+RCP		+XCP		+CONGA	
Pkt Hdr Vector	187	191	+2%	195	+4%	199	+6%
Pipeline Stages	9	10	+11%	9	+0%	11	+22%
Match Crossbar	462	473	+2%	471	+2%	478	+3%
Hash Bits	1050	1115	+6%	1058	+1%	1137	+8%
SRAM	165	175	+6%	172	+4%	213	+29%
TCAM	43	44	+2%	45	+5%	44	+2%
ALU Instruction	83	88	+6%	92	+11%	98	+18%

Table 4: Summary of resource usage for various use-cases.

We conclude that RCP can indeed be implemented with adequate accuracy and limited additional resource usage. This gives us confidence that other resource allocation algorithms might be implementable as well and we do so in the following subsections.

4.4 XCP

Like RCP, the eXplicit Control Protocol (XCP) [20] is a congestion control system that relies on explicit feedback from the network, but optimizes fairness and efficiency over high bandwidth-delay links. An XCP-capable router maintains two control algorithms that are executed periodically on each output port: a congestion controller and a fairness controller. The congestion controller is similar to RCP's—it computes the desired increase or decrease in the aggregate traffic (in bytes) over the next control interval as $\phi = \alpha \cdot d \cdot S - \beta \cdot Q$, where S, Q, and d are defined as before.

The fairness controller distributes the aggregate feedback ϕ among individual packets to achieve per-flow fairness. It uses the same additive increase, multiplica-

tive decrease (AIMD) [7] principle as TCP. If $\phi > 0$, it increases the throughput of all flows by the same uniform amount. If $\phi < 0$, it decreases the throughput of a flow by a value proportional to the flow's current throughput. XCP achieves AIMD control without requiring per-flow state by sending feedback in terms of change in congestion window, and through a formulation of the feedback values designed to normalize feedback to account for variations in flow rates, packet sizes, and RTTs.

In particular, given a packet i of size s_i corresponding to a flow with current congestion window $cwnd_i$ and RTT rtt_i, XCP computes the positive feedback p_i (when $\phi > 0$) and the negative feedback n_i (when $\phi < 0$) as:

$$p_i = \xi_p \cdot \frac{rtt_i^2 \cdot s_i}{cwnd_i} \qquad n_i = \xi_n \cdot rtt_i \cdot s_i$$

where ξ_p and ξ_n are constants for a given control interval. Observe that the negative feedback is simply a uniform constant across packets if all flows have the same RTT and send the same sized packets. As a consequence, flows that send more packets (and hence operate at a higher rate) will get a proportionally higher negative feedback and will multiplicatively decrease their sending rates in the next control interval. Similarly, the structure of p_i results in an additive increase as the per-packet feedback is inversely proportional to $cwnd_i$. Finally, the per-interval constants ξ_p and ξ_n are computed such that the aggregate feedback provided across all packets equals ϕ, with L being the set of packets seen by the router in the control interval:

$$\xi_p = \frac{\phi}{d \cdot \sum_L \frac{rtt_i \cdot s_i}{cwnd_i}} \qquad \xi_n = \frac{-\phi}{d \cdot \sum_L s_i}$$

FlexSwitch Implementation

The core of the XCP protocol requires each switch to: (i) calculate and store at every control interval the values ϕ, ξ_p, and ξ_n, and (ii) compute for every packet the positive or negative feedback values (p_i or n_i). p_i and n_i are communicated back to the sender, while $cwnd_i$ and rtt_i are communicated to other routers to allow them to compute ξ_p and ξ_n. Given the programmable parser in a FlexSwitch, it is straightforward to extend the packet header to include fields corresponding to $cwnd_i$, rtt_i, and either p_i or n_i. However, the XCP equations described above require complex calculations. We outline a sequence of refinements designed to address this.

Approximate computations: As with RCP, we make use of the *metering utilization and queueing* building block and then suitably choose the stability constants and the control interval period to simplify computations in the processing pipeline. First, we set the stability constants α, β to be negative powers of 2, such that ϕ can be calculated using bit shifts. XCP is stable for $0 < \alpha < \frac{\pi}{4\sqrt{2}}$ and $\beta = \alpha^2\sqrt{2}$, which makes this simplification feasible.

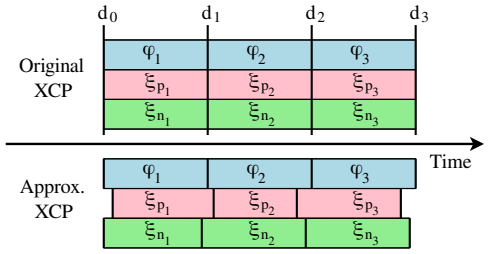


Figure 4: Staggered evaluation of XCP control parameters.

Next we approximate the control interval d to be a power of two that approximates the average RTT in the datacenter. We can then compute ϕ/d every d microseconds using integer counters and bit-shifts without incurring the loss in precision associated with the approximate division building block.

End-host computations: In XCP, end-hosts send rtt_i and $cwnd_i$ values with every packet and receive from the switches a per-packet feedback p_i, n_i. In our FlexSwitch implementation, we offload more of the computation to the end-hosts. First, we require the end-host to send to switches the computed value $\frac{rtt_i \cdot s_i}{cwnd_i}$. Second, instead of calculating the per-packet feedback p_i, n_i at the switch, we simply send the positive and negative feedback coefficients ξ_p, ξ_n to the end-host. The end-host can then calculate the necessary feedback by computing p_i, n_i based on its local flow state. This results in a subtle approximation as we can no longer keep track of aggregate feedback at the router. It is possible that we give out more aggregate feedback than the target ϕ if a large burst of packets arrives during a control interval. This can temporarily cause a queue buildup inside the switch, but XCP's negative feedback will quickly dissipate the queue. We did not see large queue buildups in our simulations.

Approximate control intervals: Finally, instead of computing the positive and negative feedback coefficients ξ_p, ξ_n along with ϕ at the end of every control interval, we compute them whenever the denominator reaches a value close to a power of 2. In our case, we approximate $\sum_L \frac{rtt_i \cdot s_i}{cwnd_i}$ or $\sum_L s_i$ to a power of 2 for positive and negative feedback coefficients respectively, both of which are in the XCP congestion header and are accumulated in integer counters inside switch memory. As a result, calculating ξ_p, ξ_n requires only a bit-shift operation. It also means that we are not calculating all XCP parameters synchronously at every control interval, but rather at staggered intervals as shown in Figure 4.

We again measure the impact of approximation on the accuracy of our FlexSwitch implementation by comparing it (XCP-Approx) to an original implementation of XCP using the simulated workload described in Section 4.1. From Figure 3 we can see that XCP-Approx also closely matches the performance of XCP, for all three types of traffic flows. Both implementations out-

perform RCP for long flows and perform worse than RCP for short flows. This validates our simulation as XCP is optimized for long flows over high bandwidth-delay links, while RCP is optimized for short flows.

Table 4 shows the additional hardware resources used by XCP-Approx when integrated into the baseline switch. XCP requires almost twice the computational resources of RCP—both in terms of ALU instructions and state carried among pipeline stages in the packet header vector. This is expected, as XCP-Approx computes 2 counter values for every packet and 3 parameters every control interval, while RCP carries out only 1 computation per interval and 1 per packet arrival. Conversely, SRAM use is diminished versus RCP, as we can carry out multiplication/division solely via bit-shifts, while we require more TCAM entries to identify when a variable is an approximate power of 2.

We conclude that XCP can also be implemented with adequate accuracy and limited additional resource usage using our building blocks. Given this experience, we now turn to a slightly broader resource management algorithm that combines some of the functionality required for RCP and XCP with a load balancing element.

4.5 CONGA

CONGA [3] is a congestion-aware load balancing system that splits TCP flows into flowlets and allocates them to paths based on congestion feedback from remote switches. Congestion is communicated among switches with each payload packet by embedding congestion information within unused bits of the VXLAN [21] overlay network header that is common in datacenters today.

CONGA operates primarily at leaf switches and does load balancing based on per-uplink congestion. Each leaf switch holds two tables that contain congestion information along all possible uplink choices from and to other leaf switches and are updated by information from these other leaf switches. To forward a packet, a leaf switch picks an outgoing link and records its choice in the packet header. Core switches record the maximum congestion along the path to the packet's destination by updating a header field with the maximum of their local congestion and that already in the field. Finally, the destination leaf switch updates its congestion-from-leaf table according to the recorded congestion along the sender's uplink port choice. To relay congestion information back to sender switches, each switch additionally chooses one entry in its congestion-from-leaf table for that switch and transmits it using another set of CONGA header fields within each payload packet.

To estimate local link congestion, all switches use a discounting rate estimator (DRE). DREs use a single register X to keep track of bytes sent along a link over a window of time. For each sent packet, the register is incremented by that packet's size. The register is decremented periodically with a multiplicative factor α between 0 and 1: $X \leftarrow X \times (1-\alpha)$.

CONGA's load-balancing algorithm is triggered for each new flowlet. It first computes for each uplink the maximum of the locally measured congestion of the uplink and the congestion feedback received for the path from the destination switch. It then chooses the uplink with the minimum congestion for the flowlet.

To recognize TCP flowlets, leaf switches keep flowlet tables. Each table entry contains an uplink number, a valid bit, and an age bit. For each incoming packet, the corresponding flowlet is determined via a hash on the packet's connection 5-tuple that indexes into the table. If the flowlet is valid (valid bit set), we simply forward the packet on the recorded port. If it is not valid, we set the valid bit and determine the uplink port via the load-balancing algorithm. Each incoming packet additionally resets the age bit of its corresponding flowlet table entry. A timer periodically checks the age bit of all flowlets before setting it. If a timer encounters an already set age bit, then the flowlet is timed out by resetting the valid bit. To accurately detect enough flowlets, CONGA switches have on the order of 64K flowlet entries in a table.

FlexSwitch Implementation

Relaying CONGA congestion information is straightforward in FlexSwitches: We simply add the required fields to the VXLAN headers. Since CONGA is intended to scale only within 2-layer cluster sub-topologies, congestion tables are also small enough to be stored entirely within FlexSwitches. To implement the remaining bits of CONGA, we need the following building blocks:

Measuring Sending Rates: We need to measure the rate of bytes sent on a particular port, which CONGA implements with DREs. We use the timer-based rate measurement building block for this task. We setup one block for each egress port to track the number of transmitted bytes along that port and set the timeout and multiplicative decrease factor to CONGA-specific values.

Maintaining congestion information: We can simply fix the size of congestion tables to a power of 2 and use precise multiplication via bit shifts and addition to index into the 2-dimensional congestion tables.

Flowlet detection: To identify TCP flowlets and remember their associated paths we use the flow statistics building block. We mimic CONGA's flowlet switching without a need for valid or age bits.

CONGA flowlet switching: To remember the initially chosen uplink for each flowlet, we store the uplink number as a tag inside the high-order bits of the flow's associated counter. We use this tag directly for routing. At the same time, we ensure that per-flow counters are reset fre-

quently enough so that the tag value will never be overwritten due to the flow packet counter growing too large. To do so, we simply calculate the maximum number of min-sized packets that can traverse an uplink within a given time frame and ensure that this number stays below the allocated counter size (32 bits), minus the space reserved for the tag (2 bits in current ToR switches). For 100Gb/s uplinks, more than 5 seconds may pass sending min-sized packets at line-rate before counter overflow.

CONGA's load balancing requires a complex calculation of minimums and maximums—too many to be realized efficiently on a FlexSwitch. Rather than faithfully replicating the algorithm and determining the best uplink port upon each new flowlet, we keep a running tally of the minimally-congested uplink choice for each leaf switch. Upon a new flowlet, we simply forward along the corresponding minimally-congested uplink.

Our running tally needs to be updated each time a corresponding congestion metric is updated. If this results in a new minimum, we simply update the running minimum. However, if an update causes our current minimum to fall out of favor, we might need to find the new current minimum. This would require re-computation of the current congestion metric for all possibilities—an operation we want to avoid. Instead, we simply update our current "minimum" to the value we have at hand as a best guess and wait for further congestion updates from remote switches to tell us the true new minimum. In our current implementation we also do not update our tally when local rate measurements are updated. This spares further resources at minimal accuracy loss. We ran ns3 simulations comparing CONGA and approximated CONGA implemented using our building blocks to confirm that our approximations don't cause significant deviation in performance (see Appendix B).

Table 4 shows the resources used when our CONGA implementation is added to our baseline switch implementation. We can see that an additional 29% of SRAM to store the additional congestion and flowlet tables is the main contribution. Also, we require 2 extra pipeline stages and 18% more ALU instructions to realize CONGA computationally, for example to approximate multiplication and division. Other resource use is increased minimally, below 10%.

We conclude that even complex load balancing protocols can be implemented on FlexSwitches using our building blocks. The additional resource use is moderate, taking up less than a third of the baseline amount of stages, SRAM bytes, and ALU instructions and less than 10% of the baseline for the other switch resources.

5 Related Work

Our work demonstrates that many resource allocations protocols can be implemented on a FlexSwitch using various approximation techniques. To achieve our goal, we leverage several algorithms from the streaming literature and apply them to a switch setting. For example, we show how to implement the HyperLogLog [28] algorithm and count-min sketch [16] on a FlexSwitch to approximate the number and frequency of distinct elements in traffic flows. Our flow timestamps and flowlet detection building blocks are related to *Approximate Concurrent State Machines* [11], but we are able to design simpler solutions given that we don't need general state machines to implement the functionality. Other related efforts are OpenSketch [38] and DREAM [23] that propose software-defined measurement architectures. OpenSketch uses sketches implemented on top of NetFPGA, while DREAM centralizes the heavy-hitter detection at a controller while distributing the flow monitoring tasks over multiple traditional OpenFlow switches. Both works trade off accuracy for resource conservation. We build on these ideas to implement a broad set of building blocks within the constraints of the hardware model and implement resource allocation algorithms using them.

Other work has proposed building blocks to aid programmable packet scheduling [35] and switch queue management [36]. While some of this work has been implemented on NetFPGA, we believe their solutions are likely to be applicable within a FlexSwitch model albeit with some approximations. Complementary to our work are proposals for enhancing the programmability of a reconfigurable pipeline [33].

6 Conclusion

As switching technology evolves to provide flexible M+A processing for each forwarded packet, it holds the promise of making a software-defined dataplane a reality. This paper evaluates the power of flexible M+A processing for in-network resource allocation. While hardware constraints make the implementation of resource allocation algorithms difficult, we show that it is possible to approximate several popular algorithms with acceptable accuracy. We develop a number of building blocks that are broadly applicable to a wide range of network allocation and management applications. We use implementations on a production FlexSwitch, compilation statistics regarding hardware resources allocated, and simulation results to show that our protocol implementations are feasible and effective.

Acknowledgments

We would like to thank the anonymous NSDI reviewers and our shepherd Rodrigo Fonseca for their valuable feedback. This research was partially supported by the National Science Foundation under Grants CNS-1518702 and CNS-1616774.

References

[1] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (2010).

[2] ALIZADEH, M., ATIKOGLU, B., KABBANI, A., LAKSHMIKANTHA, A., PAN, R., PRABHAKAR, B., AND SEAMAN, M. Data Center Transport Mechanisms: Congestion Control Theory and IEEE standardization. In *Proceedings of the 46th Annual Allerton Conference on Communication, Control, and Computing* (2008).

[3] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., VAIDYANATHAN, R., CHU, K., FINGERHUT, A., LAM, V. T., MATUS, F., PAN, R., YADAV, N., AND VARGHESE, G. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *Proceedings of the ACM SIGCOMM Conference* (2014).

[4] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM Conference* (2010).

[5] ALIZADEH, M., KABBANI, A., EDSALL, T., PRABHAKAR, B., VAHDAT, A., AND YASUDA, M. Less Is More: Trading a Little Bandwidth for ultra-Low Latency in the Data Center. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012).

[6] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pFabric: Minimal Near-optimal Datacenter Transport. In *Proceedings of the ACM SIGCOMM Conference* (2013).

[7] ALLMAN, M., PAXSON, V., AND BLANTON, E. TCP Congestion Control. RFC 5681, 2009.

[8] AZAR, Y., BRODER, A. Z., KARLIN, A. R., AND UPFAL, E. Balanced Allocations. *SIAM Journal on Computing 29*, 1 (1999), 180–200.

[9] BAI, W., CHEN, L., CHEN, K., HAN, D., TIAN, C., AND WANG, H. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *12th USENIX Symposium on Networked Systems Design and Implementation* (2015).

[10] BAREFOOT NETWORKS. Tofino Programmable Switch. `https://www.barefootnetworks.com/technology/`.

[11] BONOMI, F., MITZENMACHER, M., PANIGRAH, R., SINGH, S., AND VARGHESE, G. Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines. In *Proceedings of the ACM SIGCOMM Conference* (2006).

[12] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Computer Communication Review 44*, 3 (July 2014), 87–95.

[13] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM Conference* (2013).

[14] CAVIUM. CNX880XX_PB_p1 Rev1 - Cavium. `http://www.cavium.com/pdfFiles/CNX880XX_PB_Rev1.pdf`.

[15] CAVIUM. XPliant Ethernet Switch Product Family. `http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html`.

[16] CORMODE, G., AND MUTHUKRISHNAN, S. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms 55*, 1 (2005), 58–75.

[17] DUKKIPATI, N. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. PhD thesis, Stanford University, Dept. of Electrical Engineering, 2007.

[18] HE, K., ROZNER, E., AGARWAL, K., FELTER, W., CARTER, J., AND AKELLA, A. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *Proceedings of the ACM SIGCOMM Conference* (2015).

[19] JOSE, L., YAN, L., VARGHESE, G., AND MCKEOWN, N. Compiling Packet Programs to Reconfigurable Switches. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (2015).

[20] KATABI, D., HANDLEY, M., AND ROHRS, C. Congestion Control for High Bandwidth-delay Product Networks. In *Proceedings of the ACM SIGCOMM Conference* (2002).

[21] MAHALINGAM, M., DUTT, D., DUDA, K., AGARWAL, P., KREEGER, L., SRIDHAR, T., BURSELL, M., AND WRIGHT, C. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. RFC 7348, 2014.

[22] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review 38*, 2 (Mar. 2008), 69–74.

[23] MOSHREF, M., YU, M., GOVINDAN, R., AND VAHDAT, A. DREAM: Dynamic Resource Allocation for Software-defined Measurement. In *Proceedings of the ACM SIGCOMM Conference* (2014).

[24] NICHOLS, K., AND JACOBSON, V. Controlling Queue Delay. *Queue 10*, 5 (May 2012).

[25] NS-3 CONSORTIUM. ns-3 Network Simulator. `http://www.nsnam.org/`.

[26] OZDAG, R. Intel® Ethernet Switch FM6000 Series-Software Defined Networking. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf.

[27] PATEL, P., BANSAL, D., YUAN, L., MURTHY, A., GREENBERG, A., MALTZ, D. A., KERN, R., KUMAR, H., ZIKOS, M., WU, H., KIM, C., AND KARRI, N. Ananta: Cloud Scale Load Balancing. In *Proceedings of the ACM SIGCOMM Conference* (2013).

[28] PHILIPPE FLAJOLET AND ÉRIC FUSY AND OLIVIER GANDOUET AND FRÉDÉRIC MEUNIER. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *HAL CCSD; Discrete Mathematics and Theoretical Computer Science* (June 2007), 137–156.

[29] POPA, L., KRISHNAMURTHY, A., RATNASAMY, S., AND STOICA, I. FairCloud: Sharing the Network in Cloud Computing. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks* (2011).

[30] ROESCH, M. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration* (1999).

[31] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the Social Network's (Datacenter) Network. In *Proceedings of the ACM SIGCOMM Conference* (2015).

[32] SHIEH, A., KANDULA, S., GREENBERG, A., KIM, C., AND SAHA, B. Sharing the Data Center Network. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (2011).

[33] SIVARAMAN, A., CHEUNG, A., BUDIU, M., KIM, C., ALIZADEH, M., BALAKRISHNAN, H., VARGHESE, G., MCKEOWN, N., AND LICKING, S. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the ACM SIGCOMM Conference* (2016).

[34] SIVARAMAN, A., KIM, C., KRISHNAMOORTHY, R., DIXIT, A., AND BUDIU, M. DC.P4: Programming the Forwarding Plane of a Data-center Switch. In *Proceedings of the ACM SIGCOMM Symposium on Software Defined Networking Research* (2015).

[35] SIVARAMAN, A., SUBRAMANIAN, S., ALIZADEH, M., CHOLE, S., CHUANG, S.-T., AGRAWAL, A., BALAKRISHNAN, H., EDSALL, T., KATTI, S., AND MCKEOWN, N. Programmable Packet Scheduling at Line Rate. In *Proceedings of the ACM SIGCOMM Conference* (2016).

[36] SIVARAMAN, A., WINSTEIN, K., SUBRAMANIAN, S., AND BALAKRISHNAN, H. No Silver Bullet: Extending SDN to the Data Plane. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks* (2013).

[37] WILSON, C., BALLANI, H., KARAGIANNIS, T., AND ROWTRON, A. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM Conference* (2011).

[38] YU, M., JOSE, L., AND MIAO, R. Software Defined Traffic Measurement with OpenSketch. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013).

[39] ZHOU, J., TEWARI, M., ZHU, M., KABBANI, A., POUTIEVSKI, L., SINGH, A., AND VAHDAT, A. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *Proceedings of the Ninth European Conference on Computer Systems* (2014).

A Building Blocks and their Use in Various Protocols

Building Block	Functionality	Techniques
Cardinality Estimator	Estimate #unique elements in a stream	Linear Counting, Hyperloglog
Flow Counters	Estimate per-flow bytes/packets	Count-min sketch
Flow Timestamps	Last packet sent timestamp	Sketch with timestamps
Metering queues/utilization	Estimate rates	EWMA
Port Balancing	Pick a port based on some metric	Power-of-2 choices
Approx Arithmetic	Division/Multiplication	Logarithm tables, Bit-shits

Table 5: Summary of the proposed building blocks and techniques we use to implement them.

Functionality	Protocol	Building Blocks Required	Implementation
Congestion Control, Scheduling	RCP [17]	Arithmetic, Cardinality, Metering	Section 4.3.
	XCP [20]	Arithmetic, Metering	Section 4.4.
	QCN [2]	Arithmetic, Metering	Meter queue sizes and calculate feedback value from switch queue length based on sampling probability.
	HULL [5]	Arithmetic, Metering	Implement *Phantom queues* by metering and mark packets based on utilization levels.
	D^3 [37]	Flow Statistics, Metering	RCP like feedback, but prioritizes near-deadline flows.
	PIAS [9]	Flow Statistics, Balancing	Emulates shortest job next scheduling by dynamically lowering priority based on packets sent.
Load Balancing	CONGA [3]	Arithmetic, Flow Timestamps, Metering	Section 4.5.
	WCMP [39]	Balancing	Section 4.2.
	Ananta [27]	Flow Counters, Metering, Balancing	Use flow counters to realize VIP map and flow table. Use metering and balancing for packet rate fairness and to detect heavy hitters.
	Hedera [1]	Flow Counters, Balancing	Detect heavy hitters and balance them.
	Presto [18]	Flow Statistics, Balancing	Create flowlets and balance them
QoS & Fairness	Seawall [32]	Arithmetic, Cardinality, Metering	Collect traffic statistics and provide RCP-like feedback to determine the per-link, per-entity share.
	FairCloud [29]	Arithmetic, Flow Counters	Meter number of source/destination flows and queue them into approximately prioritized queues.
	CoDel [24]	Arithmetic, Metering	Section 4.2.
	pFabric [6]	Arithmetic, Flow Counters	Queue packets into approximately prioritized queues using remaining flow length value in the header.
Access Control	Snort IDS [30]	Flow Counters, Cardinality	Section 4.2.
	OpenSketch [38]	Flow Counters, Metering	Our approximate flow statistics are based on the same underlying count-min sketch.

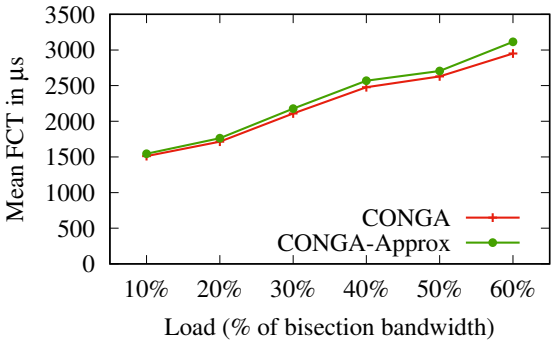
Table 6: Network functions that can be supported using FlexSwitch building blocks. A deeper discussion of several of the algorithms is given in Section 4.

```
sketch = {ts, route_id}[2][N]

elements(five_tuple):
  h1, h2 = hashes(five_tuple)
  e1 = sketch[0][h1 % N]
  e2 = sketch[1][h2 % N]

read(five_tuple):
  e1, e2 = elements(five_tuple)
  ts = min(e1.ts, e2.ts)
  route = e1.route_id + e2.route_id
  return (ts, route)
```

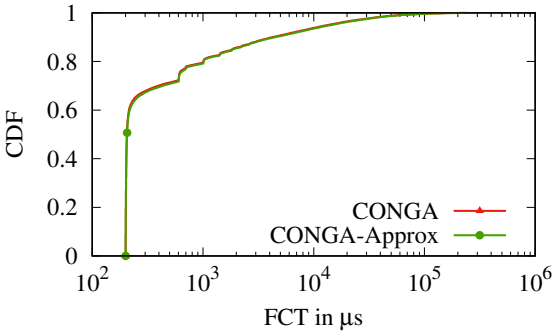
```
update(five_tuple, ts, route_id):
  e1, e2 = elements(five_tuple)
  cutoff = ts - TIMEOUT
  if (e1.ts < cutoff && e2.ts < cutoff)
    e1.route_id = rand()
    e2.route_id = route_id - e1.route_id
  else if (e1.ts < cutoff)
    e1.route_id = route_id - e2.route_id
  else if (e2.ts < cutoff)
    e2.route_id = route_id - e1.route_id

  e1.ts = e2.ts = ts
```

Figure 5: Pseudocode for the flowlet switching building block presented in Section 3.1. This particular sketch uses two hash functions and two rows of N counters each, but generalizes to multiple hash functions in a straightforward manner.

B Approximate CONGA ns3 simulation



(a) Average FCT with increasing network load.



(b) CDF of FCT with network load at 60%.

Figure 6: Performance comparison between CONGA and approximate CONGA using ns3 simulations.

We compare the performance of approximate CONGA implemented using our building blocks to the original version of CONGA. We simulate the same 4-switch (2 leaves, 2 spines), 64 server topology in [3] with $50\mu s$ link latency and the default CONGA parameters: $Q = 3, \tau = 160\mu s$, and flowlet gap $T_{fl} = 500\mu s$. We run the enterprise workload described in the same paper, and vary the arrival rate to achieve a target network load, which we measure as a percentage of the total bisection bandwidth.

First, we measure the change in average flow completion time as we increase the load in the network. Figure 6a shows that our approximate implementation closely follows the original protocol. We sometimes see a slightly higher FCT primarily because we perform an approximate *minimum* over the ports when we have to assign a new flowlet. Current restrictions on FlexSwitches don't allow us to scan all ports to pick the least loaded one, so we keep a running approximate minimum. This results in some flowlets not getting placed optimally to the least loaded link. In all other cases, we implement CONGA's protocols accurately.

Figure 6b shows the CDF of flow completion times for all flows when the network is loaded at 60%. Again, the approximate implementation of CONGA matches very closely to that of original CONGA. A majority of the flows are short and hence are not affected by the approximate minimum selection of ports.

APUNet: Revitalizing GPU as Packet Processing Accelerator

Younghwan Go, Muhammad Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park

School of Electrical Engineering, KAIST

Abstract

Many research works have recently experimented with GPU to accelerate packet processing in network applications. Most works have shown that GPU brings a significant performance boost when it is compared to the CPU-only approach, thanks to its highly-parallel computation capacity and large memory bandwidth. However, a recent work argues that for many applications, the key enabler for high performance is the inherent feature of GPU that automatically hides memory access latency rather than its parallel computation power. It also claims that CPU can outperform or achieve a similar performance as GPU if its code is re-arranged to run concurrently with memory access, employing optimization techniques such as group prefetching and software pipelining.

In this paper, we revisit the claim of the work and see if it can be generalized to a large class of network applications. Our findings with eight popular algorithms widely used in network applications show that (a) there are many compute-bound algorithms that do benefit from the parallel computation capacity of GPU while CPU-based optimizations fail to help, and (b) the relative performance advantage of CPU over GPU in most applications is due to data transfer bottleneck in PCIe communication of discrete GPU rather than lack of capacity of GPU itself. To avoid the PCIe bottleneck, we suggest employing integrated GPU in recent APU platforms as a cost-effective packet processing accelerator. We address a number of practical issues in fully exploiting the capacity of APU and show that network applications based on APU achieve multi-10 Gbps performance for many compute/memory-intensive algorithms.

1 Introduction

Modern graphics processing units (GPUs) are widely used to accelerate many compute- and memory-intensive applications. With hundreds to thousands of processing cores and large memory bandwidth, GPU promises a great potential to improve the throughput of parallel applications. Fortunately, many network applications fall into this category as they nicely fit the execution model of GPU. In fact, a number of research works [29–31, 33, 34, 46, 47] have reported that GPU helps improve the performance of network applications.

More recently, however, the relative capacity of GPU over CPU in accelerating network applications has been questioned. A recent work claims that most benefits of GPU come from fast hardware thread switching that transparently hides memory access latency instead of its high computational power [32]. They have also shown that applying the optimization techniques of group prefetching and software pipelining to CPU code often makes it more resource-efficient than the GPU-accelerated version for many network applications.

In this paper, we re-examine the recent claim on the efficacy of GPU in accelerating network applications. Through careful experiments and reasoning, we make following observations. First, we find that the computational power of GPU does matter in improving the performance of many compute-intensive algorithms widely employed in network applications. We show that popular cryptographic algorithms in SSL and IPsec such as RSA, SHA-1/SHA-2, and ChaCha20 highly benefit from parallel computation cycles rather than transparent memory access hiding. They outperform optimized CPU code by a factor of 1.3 to 4.8. Second, while the CPU-based code optimization technique like G-Opt [32] does improve the performance of naïve CPU code, its acceleration power is limited when it is compared with that of GPU without data transfer. By the performance-per-cost metric, we find that GPU is 3.1x to 4.8x more cost-effective if GPU can avoid data transfer overhead. Third, we emphasize that the main bottleneck in GPU workload offloading lies in data transfer overhead incurred by the PCIe communication instead of lack of capacity in the GPU device itself. However, masking the DMA delay is challenging because individual packet processing typically does not require lots of computation or memory access. This makes it diffi-

cult for asynchronous data transfer (e.g., concurrent copy and execution of GPU kernel) to completely overlap with GPU kernel execution. Given that typical PCIe bandwidth is an order of magnitude smaller than the memory bandwidth of GPU, it is not surprising that data transfer on PCIe limits the performance benefit in GPU-accelerated network applications.

To maximize the benefit of GPU without data transfer overhead, we explore employing integrated GPU in Accelerated Processing Units (APUs) for network applications. APU is attractive in packet processing as it provides a single unified memory space for both CPU and GPU, eliminating the data transfer overhead in discrete GPU. In addition, its power consumption is only a fraction of that of typical CPU (e.g., 35W for AMD Carrizo) with a much smaller price tag (e.g., $100 to $150), making it a cost-effective accelerator for networked systems. The question lies in how we can exploit it as a high-performance packet processing engine.

However, achieving high performance in packet processing with APU faces a few practical challenges. First, in contrast to discrete GPU, integrated GPU loses the benefit of high-bandwidth GDDR memory as it has to share the DRAM with CPU. Since both CPU and GPU contend for the shared memory bus and controller, efficient use of memory bandwidth is critical for high performance. Second, the communication overhead between CPU and GPU to switch contexts and to synchronize data update takes up a large portion of GPU execution time. Unlike discrete GPU, this overhead is no longer masked by overlapped GPU kernel execution. Also, cache coherency in APU is explicitly enforced through expensive instructions since CPU and GPU employ a separate cache. This implies that the results by GPU threads are not readily visible to CPU threads without heavyweight synchronization operations or slow GPU kernel teardown.

We address these challenges in a system called APUNet, a high-performance APU-accelerated network packet processor. For efficient utilization of memory bandwidth, APUNet extensively exercises zero-copying in all stages of packet processing: packet reception, processing by CPU and GPU, and packet transmission. We find that the extensive zero-copying model helps extract more computational power from integrated GPU, outperforming the naïve version by a factor of 4.2 to 5.4 in IPsec. For low-latency communication between CPU and GPU, APUNet adopts persistent GPU kernel execution that keeps GPU threads running in parallel across a continuous input packet stream. Eliminating GPU kernel launch and teardown, combined with zero copying significantly reduces packet processing latency by 7.3x to 8.2x and waiting time to collect a batch of packets for parallel execution. In addition, APUNet performs "group synchronization" where a batch of GPU threads implicitly synchronize the memory region of the processed packets, which avoids heavy atomics instructions. It efficiently ensures coherent data access between CPU and GPU by checking for updated packet payload (e.g., non-zero values in HMAC digest) only when processing is completed, which improves the synchronization performance by a factor of 5.7.

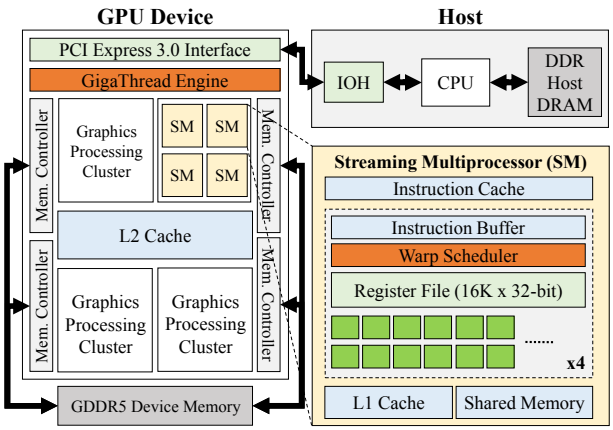


Figure 1: Typical architecture of discrete GPU

APUNet achieves both high-speed and cost-efficient network packet processing on several popular network applications. We find that many network applications, regardless of being memory- or compute-intensive, outperform CPU baseline as well as optimized implementations. APUNet improves G-Opt's performance by up to 1.6x in hashing-based IPv6 forwarding, 2.2x in IPsec gateway, 2.8x in SSL proxy, and up to 4.0x in NIDS adopting the Aho-Corasick pattern matching algorithm. We note that APUNet does not improve the performance of simple applications like IPv4 forwarding as the offloaded GPU kernel task is too small to overcome the CPU-GPU communication overhead.

2 Background

In this section, we describe the architecture of discrete and integrated GPUs and analyze their differences.

2.1 Discrete GPU

Most GPU-accelerated networked systems [29–31, 33, 34, 46, 47] have employed discrete GPU to enhance the performance of network applications. Typical discrete GPU takes the form of a PCIe peripheral device that communicates with the host side via PCIe lanes as shown in Figure 1. It consists of thousands of processing cores (e.g., 2048 in NVIDIA GTX980 [5]) and a separate GDDR memory with large memory bandwidth (e.g., 224 GB/s for GTX980). It adopts the single-instruction, multiple-thread (SIMT) execution model under which a group of threads (called a warp in NVIDIA or a wavefront in AMD hardware), concurrently executes the same instruction stream in a lock-step manner. Multiple warps (e.g., 4 in GTX980)

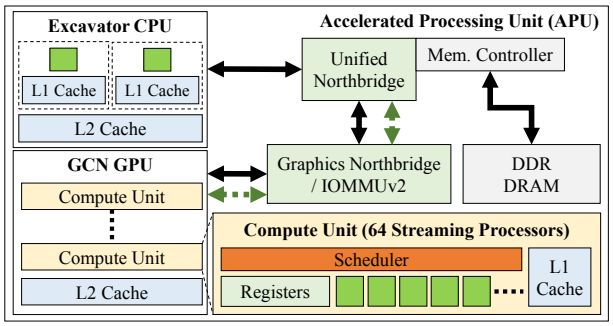


Figure 2: Architecture of integrated GPU (Carrizo APU)

are run on a streaming multiprocessor (SM) that shares caches (instruction, L1, shared memory) for fast instruction/data access and thousands of registers for fast context switching among threads. SMs are mainly optimized for instruction-level parallelism but do not implement sophisticated branch prediction or speculative execution as commonly available in modern CPU. High performance is achieved when all threads run the same basic block in parallel, but when a thread deviates from the instruction stream due to a branch, the execution of the other threads in a warp is completely serialized.

By the raw computation cycles and memory bandwidth, typical discrete GPU outperforms modern CPU. For example, NVIDIA GTX980 has 110x more computation cycles and 3.8x higher memory bandwidth than Intel Xeon E5-2650 v2 CPU. Such large computation power brings a clear performance advantage in massively-parallel workloads such as supercomputing [19, 38, 39] or training deep neural networks [21, 23, 24]. However, in other workloads that do not exhibit enough parallelism, GPU may perform poorly as it cannot fully exploit the hardware. Since our focus in this paper is packet processing in network applications, we ask a question: can GPU perform better than CPU in popular network applications? We attempt to answer this question in Section 3.

2.2 Integrated GPU

In recent years, integrated GPU has gained more popularity with extra computation power. Unlike discrete GPU, integrated GPU is manufactured into the same die as CPU and it shares the DRAM with CPU. Both Intel and AMD have a lineup of integrated GPU (e.g., AMD's Accelerated Processing Unit (APU) or Intel HD/Iris Graphics) that can be programmed in OpenCL. Figure 2 illustrates an example architecture of an integrated GPU inside AMD Carrizo APU [44].

Typical integrated GPU has lower computation capacity than discrete GPU as it has a smaller number of processing cores. For example, AMD Carrizo has 8 compute units (CUs) [1], each containing 64 streaming cores [2]. L1 cache and registers are shared among the streaming cores in a CU while L2 cache is used as a synchronization point across CUs. Despite smaller computation power, integrated GPU is still an attractive platform due to its much lower power consumption (e.g., 35W for AMD Carrizo) and a lower price tag (e.g., $100-$150 [2]). Later, we show that integrated GPU can be exploited as the most cost-effective accelerator by the performance-per-cost metric for many network applications.

The most notable aspect of integrated GPU comes from the fact that it shares the same memory subsystem with CPU. On the positive side, this allows efficient data sharing between CPU and GPU. In contrast, discrete GPU has to perform expensive DMA transactions over PCIe, which we identify as the main bottleneck for packet processing. For efficient memory sharing, OpenCL supports shared virtual memory (SVM) [9] that presents the same virtual address space for both CPU and GPU. SVM enables passing buffer pointers between CPU and GPU programs, which potentially eliminates memory copy to share the data. On the negative side, however, shared memory greatly increases the contention on the memory controller and reduces the memory bandwidth share per each processor. To alleviate the problem, integrated GPU employs a separate L1/L2 cache, but it poses another issue of cache coherency. Cache coherency across CPU and its integrated GPU can be explicitly enforced through atomics instructions of OpenCL, but explicit synchronization incurs a high overhead in practice. We address the two issues later in this paper. First, with extensive zero-copying from packet reception all the way up to GPU processing, we show that CPU and GPU share the memory bandwidth efficiently for packet processing. Second, we employ a technique that implicitly synchronizes the cache and memory access by integrated GPU to make the results of GPU processing available to CPU at a low cost.

3 CPU vs. GPU: Cost Efficiency Analysis

Earlier works have shown that GPU improves the performance of packet processing. This makes sense as network packet processing typically exhibit high parallelism. However, a recent work claims that optimized CPU code often outperforms the GPU version, with software-based memory access hiding. This poses a question: which of the two processors is more cost-effective in terms of packet processing? In this section, we attempt to answer this question by evaluating the performance for a number of popular network algorithms.

3.1 CPU-based Memory Access Hiding

Before performance evaluation, we briefly introduce the G-Opt work [32]. In that work, Kalia *et al.* have argued that the key performance contributor of GPU is not its high computation power, but its fast hardware context switch-

[1] CU is similar to NVIDIA GPU's SM.
[2] 512 cores in total vs. 2048 cores in GTX980

ing that hides memory access latency. To simulate the behavior in CPU, they develop G-Opt, a compiler-assisted tool that semi-automatically re-orders the CPU code to execute memory access concurrently without blocking. G-Opt employs software pipelining to group prefetch the data from memory during which it context switches to run other code. They demonstrate that overlapped code execution brings a significant performance boost in IPv6 forward table lookup (2.1x), IPv4 table lookup (1.6x), multi-string pattern matching (2.4x), L2 switch (1.9x), and named data networking (1.5x). They also claim that G-Opt-based CPU code is more resource-efficient than the GPU code, and show that a system without GPU offloading outperforms the same system that employs it.

In the following sections, we run experiments to verify the claims of G-Opt. First, we see if popular network algorithms do not really benefit from parallel computation cycles. This is an important question since many people believe that earlier GPU-accelerated cryptographic works [29, 31] benefit from parallel computation of GPU. We see if G-Opt provides the same benefit to compute-intensive applications. Second, we see if G-Opt makes CPU more resource-efficient than GPU in terms of memory access hiding. Given that memory access hiding of GPU is implemented in hardware, it would be surprising if software-based CPU optimization produces better performance in the feature. Note that we do not intend to downplay the G-Opt work. Actually, we do agree that G-Opt is a very nice work that provides a great value in CPU code optimization. We simply ask ourselves if there is any chance to interpret the results in a different perspective.

3.2 Experiment Setup and Algorithms

We implement a number of popular packet processing tasks for the experiments. These include the algorithms evaluated in the G-Opt paper as well as a few cryptographic algorithms widely used in network security protocols and applications.

IP forwarding table lookup: IP forwarding table lookup represents a memory-intensive workload as typical IP forwarding table does not fit into CPU cache. IPv4 table lookup requires up to two memory accesses while IPv6 table lookup requires up to seven memory lookups with hashing. We use an IPv4 forwarding table with 283K entries as in PacketShader [29] and an IPv6 table with 200K randomized entries for experiments.

Multi-string pattern matching: Aho-Corasick (AC) multi-string pattern matching [18] is one of the most popular algorithms in network intrusion detection systems (NIDS) [15, 43] and application-level firewalls [7]. AC scans each payload byte and makes a transition in a DFA table to see if the payload contains one of the string patterns. It is a memory-intensive workload as each byte requires at least five memory accesses [22].

ChaCha20-Poly1305: ChaCha20-Poly1305 [25, 26] is a relatively new cipher stream actively being adapted by a number of Web browsers (e.g., Chrome) and Google websites. We select ChaCha20-Poly1305 as it is a part of AEAD standard for TLS 1.3, making AES-GCM and ChaCha20-Poly1305 the only options for future TLS [36]. ChaCha20 [26] is a stream cipher that expands 256-bit key into 2^{64} randomly accessible streams. It is mostly compute-bound as it does not require any table lookups. Poly1305 [25], a code authenticator, is also computation-heavy as it produces a 16-byte message tag by chopping the message into 128-bit integers, adding 2^{128} to each integer, then executing arithmetic operations on each integer.

SHA-1/SHA-2: SHA-1/SHA-2 are widely used for data integrity in network security protocols such as IPsec, TLS, PGP, SSH and S/MIME. SHA-1 is being rapidly replaced by SHA-2 as the attack probability on SHA-1 increases [14]. We select one of the standards of SHA-2, SHA-256, which produces a 256-bit digest by carrying out 64 rounds of compression with many arithmetic/bit-wise logical operations. Both SHA-1/SHA-2 are compute-intensive workloads.

RSA: RSA [13] is one of the most popular public key ciphers in TLS [48]. RSA is a compute-intensive workload as it requires a large number of modular exponentiations. We use 2048-bit RSA as 1024-bit key is deprecated [12].

Test platform: For experiments, we use recent CPU and GPU (both discrete and integrated GPU) as shown in Figure 3(a). For GPU code, we either follow the efficient implementations as described in earlier works [29–31, 47] or write them from scratch. We use the latest version of OpenSSL 1.0.1f for baseline CPU code, and apply G-Opt that we downloaded from a public repository. We feed a large stream of synthetic IP packets to the algorithms without doing packet I/O.

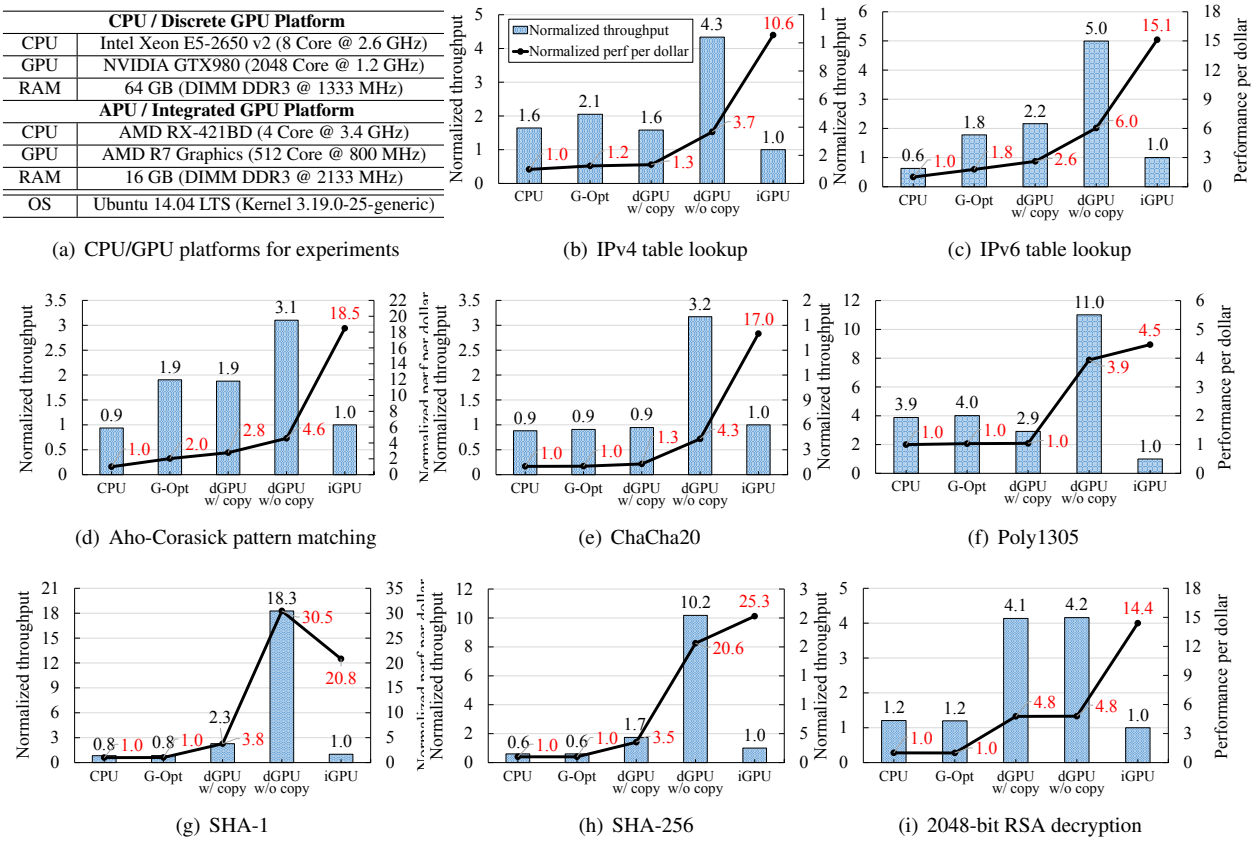
3.3 Performance Analysis

For each algorithm, we measure the throughputs of CPU baseline (CPU), G-Opt, discrete GPU with copy (dGPU w/ copy) and without copy (dGPU w/o copy), and integrated GPU (iGPU). "w/ copy" includes the PCIe data transfer overhead in discrete GPU while "w/o copy" reads the data from dGPU's own GDDR memory for processing. We include "w/o copy" numbers to figure out dGPU's inherent capacity by containing the PCIe communication overhead. Note, both "CPU" and "G-Opt" use all eight cores in the Xeon CPU while "dGPU" and "iGPU" use one CPU core for GPU kernel execution.

Performance-per-dollar metric: As each computing device has different hardware capacity with a varying price, it is challenging to compare the performance of heterogeneous devices. For fair comparison, we adopt the performance-per-dollar metric here. We draw the prices of hardware from Amazon [1] (as of September 2016)

CPU / Discrete GPU Platform	
CPU	Intel Xeon E5-2650 v2 (8 Core @ 2.6 GHz)
GPU	NVIDIA GTX980 (2048 Core @ 1.2 GHz)
RAM	64 GB (DIMM DDR3 @ 1333 MHz)
APU / Integrated GPU Platform	
CPU	AMD RX-421BD (4 Core @ 3.4 GHz)
GPU	AMD R7 Graphics (512 Core @ 800 MHz)
RAM	16 GB (DIMM DDR3 @ 2133 MHz)
OS	Ubuntu 14.04 LTS (Kernel 3.19.0-25-generic)

(a) CPU/GPU platforms for experiments



(b) IPv4 table lookup (c) IPv6 table lookup

(d) Aho-Corasick pattern matching (e) ChaCha20 (f) Poly1305

(g) SHA-1 (h) SHA-256 (i) 2048-bit RSA decryption

Figure 3: Comparison of performance-per-dollar values (line) and per-processor throughputs (bar). Performance-per-dollar values are normalized by that of CPU while per-processor throughputs are normalized by that of iGPU.

Component	Price ($)	Details
CPU	1,143.9	8 CPU cores
Discrete GPU	840.0	GPU + 1 CPU core
Integrated GPU	67.5	GPU + 1 CPU core

Table 1: Price of each processor setup

and show them in Table 1. We estimate the price of iGPU as follows. We find that Athlon 860K CPU has an almost identical CPU specification as AMD Carrizo without integrated GPU. So, we obtain the price of iGPU by subtracting the price of Athlon 860K CPU from that of AMD Carrizo (e.g., $50 = $119.9 - $69.99). For fairness, we include the price of one CPU core for GPU as it is required to run the GPU kernel. We note that the comparison by the performance-per-dollar metric should not be interpreted as evaluating the cost-effectiveness of CPU-based vs. GPU-based systems as it only compares the cost of each processor. However, we believe that it is still useful for gauging the cost-effectiveness of each processor for running network applications.

Performance comparison: Figure 3 compares the performance for each packet processing algorithm. Each line in a graph shows the performance-per-dollar values normalized by that of CPU. We use them as the main comparison metric. For reference, we also show per-device throughputs as a bar graph, normalized by that of iGPU.

We make following observations from Figure 3. First, G-Opt improves the performance of baseline CPU for memory-intensive operations (e.g., (b) to (d)) by a factor of 1.2 to 2.0. While these are slightly lower than reported by the original paper, we confirm that the optimization does take positive effect in these workloads. However, G-Opt has little impact on compute-intensive operations as shown in subfigures (e) to (i). This is because hiding memory access does not help much as computation capacity itself is the bottleneck in these workloads. Second, dGPU "w/o copy" shows the best throughputs by the raw performance. It outperforms G-Opt by a factor of 1.63 to 21.64 regardless of the workload characteristics. In contrast, the performance of dGPU "w/ copy" is mostly comparable to that of G-Opt except for compute-intensive operations. This implies that PCIe data transfer overhead is the key performance barrier for dGPU when packet contents have to be copied to dGPU memory. This is not surprising as the PCIe bandwidth of dGPU is more than an order of magnitude smaller than the memory bandwidth of dGPU. It also implies that dGPU itself has a large potential to

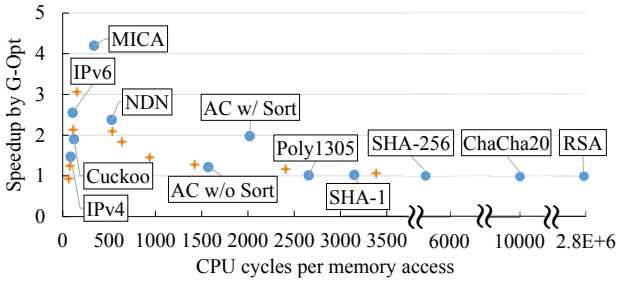


Figure 4: Correlation between speedup by G-Opt and the CPU-cycles-per-memory-access value. MICA [37] is an in-memory key-value store algorithm.

improve the performance of packet processing without the DMA overhead. Third, for compute-intensive operations, even dGPU "w/ copy" outperforms G-Opt by a large margin. By the performance-per-dollar metric, dGPU "w/-copy" brings 1.3x to 4.8x performance improvements over G-Opt for ChaCha20, SHA-1/SHA-2, and RSA. RSA shows the largest performance boost with dGPU as it transfers small data while it requires lots of computation. This confirms that compute-intensive algorithms do benefit from parallel computation cycles of GPU. Finally, and most importantly, we note that iGPU presents the best performance-per-dollar numbers in almost all cases. Even by the raw throughput, the performance of iGPU is often comparable to the baseline CPU with eight cores. This is because iGPU can fully export its computing capacity without PCIe overhead in dGPU.

Summary: We find that GPU-based packet processing not only exploits transparent memory access hiding, but also benefits from highly-parallel computation capacity. In contrast, CPU-based optimization helps accelerate memory-intensive operations, but its benefit is limited when the computation cycles are the key bottleneck. Also, dGPU has huge computation power to improve the performance of packet processing, but its improvement is curbed by the PCIe data transfer overhead. This turns our attention into iGPU as it produces high packet processing throughputs at low cost without the DMA bottleneck.

3.4 Analysis on CPU-based Optimization

We now consider how one can deterministically tell whether some packet processing task would benefit from CPU-based memory access hiding or not. Roughly speaking, if a task exhibits enough memory access that can be overlapped with useful computation, techniques like G-Opt would improve the performance. Otherwise, GPU offloading could be more beneficial if the workload can be easily parallelized.

To gain more insight, we measure the performance improvements by G-Opt for various packet processing algorithms, and correlate them with the trend of "CPU-cycles-per-memory-access", which counts the average number of CPU cycles consumed per each memory access. We measure the CPU-cycles-per-memory-access of each workload by counting the number of last-level-cache-misses of its baseline CPU code (without G-Opt optimization). For the measurement, we use Performance Application Programming Interface (PAPI) [11], a library that provides access to performance hardware counters.

Figure 4 shows the results of various algorithms. We use AMD CPU (in Figure 3(a)) for the measurement, but the similar trend is observed for Intel CPUs as well. For reference, we have a synthetic application that accesses memory at random such that we add arbitrary amount of computation between memory accesses. We mark this application as '+' in the graph. From the figure, we observe the following. First, we see performance improvement by G-Opt if x (CPU-cycles-per-memory-access) is between 50 and 1500. The best performance improvement is achieved when x is around 300 (e.g., MICA [37]). G-Opt is most beneficial if the required number of CPU cycles is slightly larger than that of memory access, reflecting a small amount of overhead to overlap computation and memory access. We think the improved cache access behavior after applying G-Opt [3] often improves the performance by more than two times. Second, we see little performance improvement if x is beyond 2500. This is not surprising since there is not enough memory access that can overlap with computation.

One algorithm that requires further analysis is Aho-Corasick pattern matching (AC w/ or w/o Sort). "AC w/o Sort" refers to the case where we apply G-Opt to the plain AC code, and we feed 1514B packets to it. "AC w/ Sort" applies extra optimizations suggested by the G-Opt paper [32] such that it pre-processes the packets with their packet header to figure out which DFA each packet falls into (called DFA number in [32]) and sorts the packets by their DFA number and packet length before performing AC. "AC w/o Sort" and "AC w/ Sort" show 1.2x and 2x performance improvements, respectively. While "AC w/o Sort" shows better performance improvement, it may be difficult to apply in practice since it needs to batch many packets (8192 packets in the experiment) and the performance improvement is subject to packet content.

4 APUNet Design and Implementation

In this section, we design and implement APUNet, an APU-accelerated network packet processing system. For high performance, APUNet exploits iGPU for parallel packet processing while it utilizes CPU for scalable packet I/O. We first describe the practical challenges in employing APU and then present the solutions that address them.

[3] For example, we see much better L1 cache hit ratio for MICA.

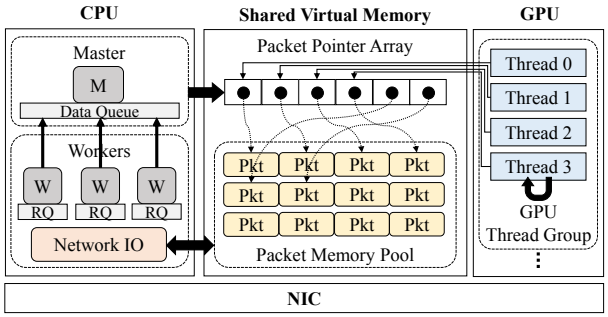


Figure 5: Overall architecture of APUNet. *M: master, W: worker, RQ: result queue.*

4.1 Challenge and Approach

APU enables efficient data sharing between CPU and GPU, but it also presents a number of practical challenges in achieving high performance. First, sharing of DRAM incurs contention on the memory controller and bus, which potentially reduces the effective memory bandwidth for each processor. Given that many packet processing tasks are memory-intensive and packet I/O itself consumes memory bandwidth, memory bandwidth is one of the key resources in network applications. Our approach is to minimize this shared memory contention by extensive zero-copy packet processing, which removes redundant data copy between CPU and GPU as well as NIC and application buffers. Second, we find that frequent GPU kernel launch and teardown incur a high overhead in packet processing. This is because GPU kernel launch and teardown in APU execute heavyweight synchronization instructions to initialize the context registers at start and to make the results visible to the CPU side at teardown. Our approach to address this problem is to eliminate the overhead with persistent thread execution. Persistent GPU thread execution launches the kernel only once, then continues processing packets in a loop without teardown. Lastly, APU does not ensure cache coherency across CPU and GPU and it requires expensive atomics operations to synchronize the data in the cache of the two processors. We address this challenge with group synchronization that implicitly writes back the cached data into shared memory only when data sharing is required. Our solution greatly reduces the cost of synchronization.

4.2 Overall System Architecture

Figure 5 shows the overall architecture of APUNet. APUNet adopts the single-master, multiple-worker framework as employed in [29]. The dedicated master exclusively communicates with GPU while the workers perform packet I/O and request packet processing with GPU via the master. The master and each worker are implemented as a thread that is affinitized to one of the CPU cores. Both CPU and GPU share a packet memory pool that stores an array of packet pointers and packet payload. They also share application-specific data structures such as an IP forwarding table and keys for cryptographic operations.

We explain how packets are processed in APUNet. APUNet reads incoming packets as a batch using the Intel DPDK packet I/O library [4]. Each worker processes a fraction of packets that are load-balanced by their flows with receive-side scaling (RSS). When a batch of packets arrive at a worker thread, the worker thread checks the validity of each packet, and enqueues the packet pointers to master's data queue. Then, the master informs GPU about availability of new packets and GPU performs packet processing with the packets. When packet processing completes, GPU synchronizes the results with the master thread. Then, the master notifies the worker thread of the results by the result queue of the worker, and the worker thread transmits the packets to the right network port. Details of data synchronization are found in the following sections.

4.3 Zero-copy Packet Processing

Zero-copy packet processing is highly desirable in APUNet for efficient utilization of the shared memory bandwidth. We apply zero-copying to all packet buffers and pass only the pointers between CPU and GPU for packet data access. In APU, one can implement zero-copying with SVM and its OpenCL API.

While SVM provides a unified virtual memory space between CPU and GPU, it requires using a separate memory allocator (e.g., `clSVMAlloc()`) instead of standard `malloc()`. However, this makes it difficult to extend zero copying to packet buffers as they are allocated in the DPDK library with a standard memory allocator. What is worse, the DPDK library allocates hugepages for packet buffers that map its virtual address space to a contiguous physical memory space. Unfortunately, DPDK does not allow turning off hugepage allocation as the rest of the code depends on the continuous physical memory space.

To address this problem, we update the DPDK code to turn off the hugepage mode, and make it use the SVM memory allocator to create packet buffers. We find that this does not result in any performance drop as address translation is not the main bottleneck in our applications. To avoid dealing with physical address mapping, we create an RDMA channel from one virtual address to another using the InfiniBand API [41] in the Mellanox NIC driver. We patch the DPDK code to support the driver in DPDK [40] and page-lock SVM-allocated buffers to be registered as the packet memory pool of the NIC. We then create an interface that exposes the SVM buffers to the application layer so that worker threads can access them to retrieve packets. While the current version depends on a particular NIC driver, we plan to extend the code to support other drivers. The total number of lines required for the modification is about 2,300 lines.

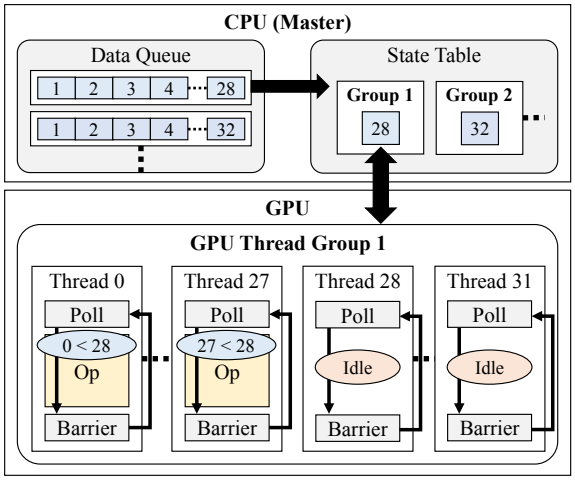


Figure 6: Persistent thread execution with a GPU thread group consisting of 32 threads

4.4 Persistent Thread Execution

GPU-based packet processing typically collects a large number of packets for high utilization of GPU threads. However, such a model comes with two drawbacks. First, it increases the packet processing delay to wait for the packets. Second, each GPU kernel launch and teardown incurs an overhead which further delays processing of the next batch of packets. The overhead is significant in APU as it has to synchronize the data between CPU and GPU at each GPU kernel launch and teardown.

To address the problem, we employ persistent GPU kernel execution. The basic idea is to keep a large number of GPU threads running to process a stream of input packets continuously. This approach has a great potential to reduce the packet processing delay as it completely avoids the overhead of kernel launch and teardown and processes packets as they are available. However, a naïve implementation could lead to serious underutilization of GPU as many GPU threads would end up executing a different path in the same instruction stream.

APUNet strives to find an appropriate balance between packet batching and parallel processing for efficient persistent thread execution. For this, we need to understand how APU hardware operates. Each CU in GPU has four SIMT units and each unit consists of 16 work items (or threads) that execute the same instruction stream in parallel [17]. So, we feed the packets to GPU so that all threads in an SIMT unit execute concurrently. At every dequeue of master's data queue, we group the packets in a multiple of the SIMT unit size, and pass them to GPU as a batch. We use 32 as the batch size in our implementation, and threads in two SIMT units (that we call as GPU thread group) process the passed packets in parallel. If the number of dequeued packets is smaller than the group size, we keep the remaining GPU threads in the group idle to make the number align with the group size. In this way, when the next batch of packets arrive, they are processed by a separate GPU thread group. We find that having idle GPU threads produces a better throughput as feeding new packets to a group of threads that are already active results in control path divergence and expensive execution serialization.

Figure 6 shows the overall architecture of persistent thread execution. CPU and GPU share a per-group *state* that identifies the mode of a GPU thread group as active (i.e., non-zero) or idle (i.e. zero). The value of the state indicates the number of packets to process. All GPU thread groups are initialized as idle and they continuously poll on the state until they are activated. When new packets arrive, the master thread finds an idle thread group and activates the group by setting the value of its group state to the number of packets to process. When the GPU thread group detects a non-zero value in its state, each GPU thread in the group checks whether it has a packet to process by comparing its local ID with the state value. Those threads whose local ID is smaller than the state value begin processing the packet inside the packet pointer array indexed by its unique global ID. Other threads stay idle waiting on a memory barrier. When all threads in the group finish execution and meet at the barrier, the thread with the minimum local ID switches the group state back to 0. The master in CPU periodically polls on the state, and when it sees that a group has completed, it retrieves the results and returns them to workers.

4.5 Group Synchronization

Another issue in persistent GPU threads lies in how to ensure cache coherency between CPU and GPU at a low cost. When GPU finishes packet processing, its result may not be readily visible in CPU as it stays in GPU's L2 cache, a synchronization point for GPU threads. In order to explicitly synchronize the update in GPU to main memory, OpenCL provides *atomics* instructions aligned with C++11 standard [16]. Atomics operations are executed through a coherent transaction path (dotted arrows in Figure 2) created by a coherent hub (CHUB) placed inside graphics northbridge [20]. Unfortunately, they are expensive operations as they require additional instructions such as locking, and CHUB can handle only one atomics operation at a time. As every atomics operation incurs an overhead, serial handling of requests from thousands of GPU threads would suffer from a significant overhead and degrade the overall performance.

We minimize the overhead by *group synchronization*. It attempts to implicitly synchronize the memory region of packets that a GPU thread group finished processing. For implicit synchronization, we exploit the LRU cache replacement policy of GPU [17] to evict dirty data items in GPU cache to main memory. We exploit idle GPU

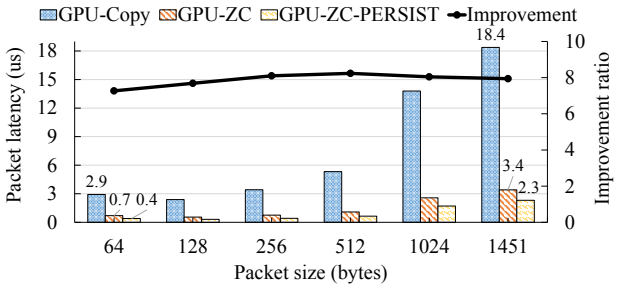


Figure 7: Packet latency with varying packet sizes

thread groups that are currently not processing packets such that they carry out dummy data I/O on a separate memory block and cause existing data in L2 cache to be evicted. While this incurs some memory contention, we find that synchronizing memory regions in a group is more efficient than synchronizing one at a time with atomics instructions. Note that idle threads (e.g., local ID larger than its group state value) in an active GPU thread group still wait on memory barrier since doing dummy data I/O will deviate them from instruction stream of active threads, making packet processing to be serialized. If all GPU thread groups are active, all GPU threads contend for GPU's cache and cause dirty data in cache to be flushed to main memory even without dummy data I/O. Finally, we guarantee data coherency between CPU and GPU by verifying the updates in packet payload (e.g., memory region for IPsec's digest contains non-zero values after processing).

4.6 Tuning and Optimizations

APUNet maximizes the performance of GPU kernel execution by applying a number of well-known optimizations, such as vectorized data accesses and arithmetic operations, minimization of thread group divergence, and loop unrolling. It also exploits GPU's high bandwidth local and private memory. We configure the number of persistent threads for maximum performance. We currently use a fixed dummy memory block size for group synchronization, and leave the design to dynamically configure the size depending on machine platform (e.g., different cache size) as a future work.

Finally, we enhance GPU's memory access speed by the hybrid usage of two SVM types: coarse- and fine-grained. Coarse-grained SVM provides faster memory access but needs explicit map and unmap operations to share the data between CPU and GPU, which incurs a high cost. Fine-grained SVM masks map/unmap overheads but shows longer memory access time due to pinned memory [27]. We thus allocate data structures that seldom change values (e.g., IP forwarding table) as coarse-grained SVM while we allocate frequently modified buffers (e.g., packet memory pool) as fine-grained SVM.

5 Evaluation

In this section, we evaluate APUNet to see how much performance improvement our design brings in packet processing. We then demonstrate the practicality of APUNet by implementing five real-world network applications and comparing the performance over the CPU-only approach.

5.1 Test Setup

We evaluate APUNet on the AMD Carrizo APU platform (shown as Integrated GPU Platform in Figure 3(a)) as a packet processing server. APUNet uses one CPU core as a master communicating with GPU while three worker cores handle packet I/O. We use a client machine equipped with an octa-core Intel Xeon E3-1285 v4 (3.50 GHz) and 32GB RAM. The client is used to either generate IP packets or HTTPS connection requests. Both server and client communicate through a dual-port 40 Gbps Mellanox ConnectX-4 NIC (MCX414A-B [3]). The maximum bandwidth of the NIC is 56 Gbps since it is using an 8-lane PCIev3 interface. We run Ubuntu 14.04 (kernel 3.19.0-25-generic) on the machines.

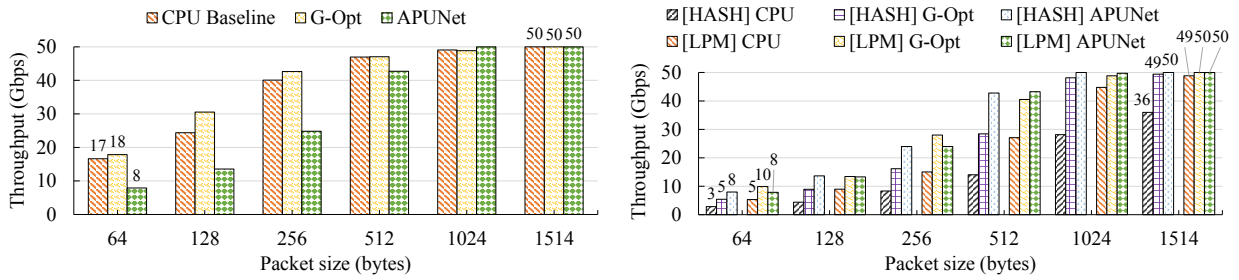
5.2 APUNet Microbenchmarks

We see if our zero-copy packet processing and persistent thread execution help lower packet processing latency, and gauge how much throughput improvement group synchronization achieves.

5.2.1 Packet Processing Latency

In this experiment, we measure the average per-packet processing latency from the time of packet arrival until its departure from the system. We evaluate the following three configurations: (i) *GPU-Copy*: a strawman GPU version with data copying between CPU and GPU address spaces (standard memory allocation in CPU with hugepages), (ii) *GPU-ZC*: GPU with zero-copy packet processing, and (iii) *GPU-ZC-PERSIST*: GPU with zero-copy packet processing *and* persistent thread execution including group synchronization. We use IPsec as packet processing task for the experiments, and use 128-bit AES in CBC mode and HMAC-SHA1 for the crypto suite.

Figure 7 compares the packet processing latency for each configuration with varying packet sizes. We set the largest packet size to 1451B, not to exceed the maximum Ethernet frame size after appending an ESP header and an HMAC-SHA1 digest. As expected, GPU-Copy shows the highest packet processing latency, which suffers from data transfer overhead between CPU and GPU. On the other hand, we see that GPU-ZC reduces the packet latency by a factor of 4.2 to 5.4 by sharing packet buffer pointers. GPU-ZC-PERSIST further reduces the latency by 1.8 times, thanks to elimination of per-batch kernel launch and teardown overheads. In total, APUNet successfully reduces the packet latency by 7.3 to 8.2 times, and shows an impressive reduction in packet processing time.



(a) IPv4 forwarding

(b) IPv6 forwarding

Figure 8: Performance of IPv4/IPv6 table lookup and packet forwarding

Moreover, it outperforms the CPU-only model by more than 2x in throughput in real-world network evaluation (Figure 9(a)).

5.2.2 Synchronization Throughput

We next evaluate the performance improvement of group synchronization when compared with synchronization with atomics operations. For this, we perform a stress test in which we pass 64B IP packets to GPU that is running an IPsec kernel code. We choose an IPsec module as our default GPU kernel since it completely updates the IP payload during encryption. Hence the entire packet content needs to be synchronized back to the CPU side. Note that zero-copy packet processing and persistent thread execution are implemented as baseline in both cases. Through measurement, we observe that APUNet achieves 5.7x throughput improvement with group synchronization (0.93 Gbps vs. 5.31 Gbps) as atomically synchronizing all dirty data serially via CHUB results in a significant performance drop. We find that APUNet significantly reduces the synchronization overhead between CPU and GPU and maintains high performance with a small cost of introducing additional memory usage from dummy data I/O.

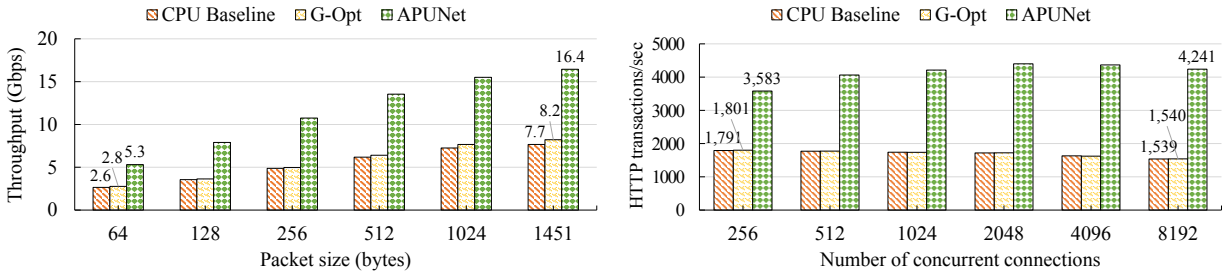
5.3 APUNet Macrobenchmarks

We determine the practicality of APUNet in real-world network applications by comparing the throughputs of CPU baseline and G-Opt enhanced implementations. We begin by briefly describing the setup of each application and then discuss the evaluation results.

IPv4/IPv6 packet forwarding: We have a client machine transmit IPv4/IPv6 packets with randomized destination IP addresses, and APUNet forwards the packets after carrying out IPv4/IPv6 table lookups. We use the same IPv4/IPv6 forwarding tables as we used for experiments in Section 3. For IPv6 table lookup, we evaluate a hashing-based lookup algorithm as well as longest prefix matching (LPM)-based algorithm. Figure 8 shows the forwarding throughputs with varying packet sizes. We see that G-Opt effectively hides memory accesses of IPv4/IPv6 table lookups, improving CPU baseline performance by 1.25x and 2.0x, respectively. Unfortunately, we find that APUNet performs worse in IPv4 forwarding due to a number of reasons. First, as IPv4 table lookup is a lightweight operation, the communication overhead between CPU and GPU takes up a relatively large portion of total packet processing time. Second, APUNet dedicates one master CPU core for communicating with GPU and employs only three CPU cores to perform packet I/O, which is insufficient to match the IPv4 forwarding performance of four CPU cores. Lastly, there is a performance penalty from the contention for a shared packet memory pool between CPU and GPU. In contrast, IPv6 packet forwarding requires more complex operations, APUNet outperforms hashing-based CPU implementations by up to 1.6*x*. We note that the LPM-based CPU implementations are resource-efficient as it does not require hashing, and its G-Opt performance is mostly comparable to that of APUNet.

IPsec gateway and SSL proxy: We now show how well APUNet performs compute-intensive applications by developing an IPsec gateway and an SSL reverse proxy. Our IPsec gateway encrypts packet payload with the 128-bit AES scheme in CBC mode and creates an authentication digest with HMAC-SHA1. Both CPU baseline and its G-Opt version of the gateway exploit Intel's AES-NI [6], which is an x86 instruction set that runs each round of AES encryption/decryption with a single AESENC/AESDEC instruction. We implement the GPU version of the IPsec gateway in APUNet by merging AES-CBC and HMAC-SHA1 code into a single unified kernel for persistent execution. Figure 9(a) shows the evaluation results. As discussed in Section 3, G-Opt fails to improve the CPU baseline performance as AES is already optimized with hardware instructions and HMAC-SHA1 operation is compute-bound. On the other hand, APUNet delivers up to 2.2*x* performance speedup, achieving 16.4 Gbps with 1451B packets.

For SSL reverse proxying, we evaluate how many SSL handshakes APUNet can perform by offloading compute-intensive RSA operations to GPU. The client machine generates encrypted SSL requests using Apache HTTP



(a) IPsec gateway

(b) SSL reverse proxy

Figure 9: Performance of compute-intensive applications. For IPsec, we use 128-bit AES in CBC mode with HMAC-SHA1. For SSL proxying, we use 2048-bit RSA, 128-bit AES in CBC mode, and HMAC-SHA1 as a ciphersuite.

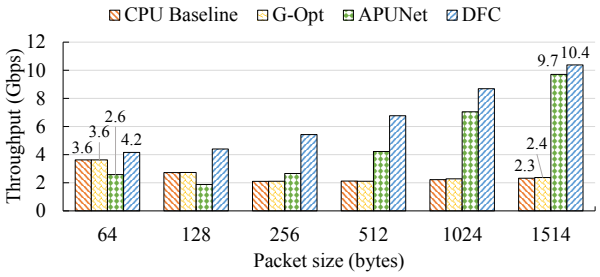


Figure 10: Performance of network intrusion detection system.

benchmark tool (`ab`) v2.3, that are then sent to APUNet. APUNet runs both SSL reverse proxy and Web server (`nginx` v1.4.6 [8]) where the proxy resides in the middle to translate between encrypted and plaintext messages. For SSL, we use TLS v1.0 with a 2048-bit RSA key, 128-bit AES in CBC mode, and HMAC-SHA1 as ciphersuite. Figure 9(b) shows the number of HTTP transactions processed per second with an increasing number of concurrent connections. Like IPsec, CPU-based optimization does not help much while our GPU-based implementation provides a performance boost for compute-intensive RSA and SHA-1 operations. Note that we have not applied persistent kernels for SSL as SSL requires switching across multiple crypto operations. Nevertheless, APUNet outperforms CPU-based approaches by 2.0x-2.8x.

Network intrusion detection system: We port a signature-based network intrusion detection system (NIDS) to APUNet. Our APUNet-based NIDS uses the Aho-Corasick (AC) pattern matching algorithm to scan ingress network traffic for known attack signatures. For evaluation, we port a full NIDS [30] that uses a Snort-based [43] attack ruleset to monitor malicious activity in a network and apply G-Opt to the AC pattern matching code for the G-Opt version. Under heavy stress, it is capable of offloading pattern matching tasks to GPU.

Figure 10 shows the throughputs against innocent synthetic traffic with randomized payload and varying packet sizes. Interestingly, we find that the full NIDS no longer benefits from G-Opt, different from 1.2x performance improvement of AC pattern matching ("AC w/o Sort") in Figure 4. This is because a fully functional NIDS would access a large number of data structures (e.g., packet acquisition, decoding and detection modules) during packet analysis and can cause data cache contention, resulting in eviction of already cached data by other prefetches before they are used. One may try to implement loop interchanging and sorting by DFA ID and packet length optimizations on all data structures as in [32] (described in Section 3.4), but this is still challenging to apply in practice as it requires batching a large number of packets (8192). In contrast, APUNet requires batching of a smaller number of packets in multiples of an SIMT unit size (e.g., 32). As a result, APUNet helps NIDS outperform AC-based CPU-only approaches by up to 4.0x for large packets. For small-sized packets, the computation load becomes small (only 10B of payload is scanned for 64B packets), making GPU offloading inefficient similar to IPv4 forwarding.

One notable thing is that DFC [22]-based NIDS outperforms APUNet -based NIDS by 7% to 61%. DFC is a CPU-based multi-string pattern matching algorithm that uses cache-friendly data structures and significantly reduces memory access at payload scanning [22]. Given that the synthetic traffic is innocent, almost all byte lookups would be hits in CPU cache as DFC uses a small table for quick inspection of innocent content. This shows that a resource-efficient algorithm often drastically improves the performance without the help of GPU. We leave our GPU-based DFC implementation as our future work.

6 Related Works

We briefly discuss previous works that accelerate network applications by optimizing CPU code or offloading the workload to GPU.

CPU code optimization: RouteBricks [28] scales the software router performance by fully exploiting the parallelism in CPU. It parallelizes packet processing with

multiple CPU cores in a single server as well as distributing the workload across a cluster of servers, and achieves 35 Gbps packet forwarding performance with four servers. DoubleClick [33] improves the performance of the Click router [35] by employing batching to packet I/O and computation. Using PSIO [10], DoubleClick batches the received packets and passes them to a user-level Click element to be processed in a group, showing 10x improvement to reach 28 Gbps for IPv4 forwarding. Similarly, we show that our APUNet prototype with recent CPU achieves up to 50 Gbps performance for IPv4/IPv6 packet forwarding. Kalia *et al.* [32] have introduced the G-Opt framework that applies memory access latency hiding to CPU code. It implements software pipelining by storing the context information in an array, then executing prefetch-and-switch to another context whenever memory access is required. Although beneficial for memory-intensive workloads, our cost-efficiency analysis shows that G-Opt is limited in improving the performance of compute-intensive applications .

GPU offloading: Most GPU-based systems have focused on exploiting discrete GPU for accelerating packet processing. PacketShader [29] is the seminal work that demonstrates a multi-10 Gbps software router by offloading workload to discrete GPU, showing close to 40 Gbps for 64B IPv4/IPv6 forwarding, and 20 Gbps for IPsec. MIDeA [47] and Kargus [30] implement GPU-based, high-performance NIDSes (10-33 Gbps). MIDeA parallelizes packet processing via multiple NIC queues and offloads pattern matching tasks to GPU whereas Kargus can offload both pattern matching and regular expression matching workloads to GPU. SSLShader [31] optimizes SSL crypto operations by exploiting GPU parallelization and demonstrates 3.7x-6.0x SSL handshake performance improvement over a CPU-only approach.

There have also been attempts to build a general GPU-based network framework. GASPP [46] integrates common network operations (e.g., flow tracking and TCP reassembly) into GPU to consolidate multiple network applications. NBA [34] extends the Click router with batched GPU processing and applies adaptive load balancing to dynamically reach near-optimal throughputs in varying environments. Although a great platform for achieving high performance, we find that discrete GPU is less cost-efficient than integrated GPU for many network applications as they suffer heavily from data transfer overhead, having RSA as a notable exception.

More recently, Tseng *et al.* [45] developed a packet processing platform using Intel GT3e integrated GPU. They employ a continuous thread design similar to our persistent thread execution but uses a shared control object supported by Intel Processor Graphic Gen8 to synchronize the data between CPU and GPU. As a result, they show a small performance improvement of 2-2.5x over 1 CPU core for IPv4/IPv6 based applications. PIPSEA [42] is another work that implements an IPsec gateway on top of APU. PIPSEA designs a packet scheduling technique that sorts the packets by their required crypto suite and packet lengths to minimize control-flow divergence. In contrast to our work, PIPSEA does not address architectural overheads of APU such as memory contention and communication overhead, showing 10.4 Gbps throughput at 1024B packet size, which is lower than 15.5 Gbps of APUNet on a similar hardware platform. To the best of our knowledge, APUNet delivers the highest performance for a number of network applications on a practical integrated GPU-based packet processor.

7 Conclusion

In this work, we have re-evaluated the effectiveness of discrete and integrated GPU in packet processing over CPU-based optimization. With the cost efficiency analysis, we have confirmed that CPU-based memory access hiding effectively helps memory-intensive network applications but it has limited impact on compute-bound algorithms that have little memory access. Moreover, we observe that relative performance advantage of CPU-based optimization is mostly due to the high data transfer overhead of discrete GPU rather than lack of its inherent capacity. To avoid the PCIe communication overhead, we have designed and implemented APUNet, an APU-based packet processing platform that allows efficient sharing of packet data. APUNet addresses practical challenges in building a high-speed software router with zero-copy packet processing, efficient persistent thread execution, and group synchronization. We have demonstrated the practicality of APUNet by showing multi-10 Gbps throughputs with low latency for a number of popular network applications. We believe that APUNet will serve as a high performance, cost-effective platform for real-world applications.

Acknowledgments

We would like to thank our shepherd Fabián Bustamante and anonymous reviewers of NSDI'17 for their insightful comments and suggestions on the paper. We also thank Anuj Kalia at CMU for in-depth discussion on experiment results and further suggestions, and Seong Hwan Kim and Dipak Bhattacharyya at AMD, Inc., for their help in understanding the internals of APU and OpenCL. This work was supported in part, by the ICT Research and Development Program of MSIP/IITP, Korea, under Grant B0101-16-1368, [Development of an NFV-inspired networked switch and an operating system for multi-middlebox services], Grant R0190-16-2012, [High Performance Big Data Analytics Platform Performance Acceleration Technologies Development], and Grant R-20160222-002755, [Cloud-based Security Intelligence Technology Development for Customized Security Service Provisioning].

References

[1] Amazon. https://www.amazon.com.

[2] AMD A10-7860K-AD786KYBJCSBX. https://www.amazon.com/AMD-Processors-Radeon-Graphics-A10-7860K-AD786KYBJCSBX/dp/B01BF377W4.

[3] ConnectX-4 Lx EN Cards. http://www.mellanox.com/page/products_dyn?product_family=219&mtag=connectx_4_lx_en_card.

[4] DPDK: Data Plane Development Kit. http://dpdk.org/.

[5] GeForce GTX 980 Specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980/specifications.

[6] Intel Advanced Encryption Standard Instruction (AES-NI). https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni.

[7] ModSecurity: Open Source Web Application Firewall. https://www.modsecurity.org.

[8] NGINX. https://www.nginx.com.

[9] OpenCL 2.0 Shared Virtual Memory Overview. https://software.intel.com/en-us/articles/opencl-20-shared-virtual-memory-overview.

[10] Packet I/O Engine. http://shader.kaist.edu/packetshader/io_engine/.

[11] Performance Application Programming Interface. http://icl.cs.utk.edu/papi/.

[12] Phasing out Certificates with 1024-bit RSA Keys. https://blog.mozilla.org/security/2014/09/08/phasing-out-certificates-with-1024-bit-rsa-keys/.

[13] Public-Key Cryptography Standards (PKCS) 1: RSA Cryptography Specifications Version 2.1. https://tools.ietf.org/html/rfc3447.

[14] Researchers Warn Against Continuing Use Of SHA-1 Crypto Standard. http://www.darkreading.com/vulnerabilities---threats/researchers-warn-against-continuing-use-of-sha-1-crypto-standard/d/d-id/1322565.

[15] Suricata Open Source IDS/IPS/NSM engine. http://suricata-ids.org/.

[16] Using OpenCL 2.0 Atomics. https://software.intel.com/en-us/articles/using-opencl-20-atomics.

[17] White Paper. AMD Graphics Core Next (GCN) Architecture. https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf.

[18] Aho, Alfred V. and Corasick, Margaret J. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, June 1975.

[19] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan. Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[20] D. Bouvier and B. Sander. Applying AMD's "Kaveri" APU for Heterogeneous Computing. https://www.hotchips.org/wp-content/uploads/hc_archives/hc26/HC26-11-day1-epub/HC26.11-2-Mobile-Processors-epub/HC26.11.220-Bouvier-Kaveri-AMD-Final.pdf.

[21] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A Flexible and Efficient Machine learning Library for Heterogeneous Distributed Systems. In *Proceedings of the NIPS Workshop on Machine Learning Systems (LearningSys)*, 2016.

[22] B. Choi, J. Chae, A. Jamshed, K. Park, and D. Han. DFC: Accelerating Pattern Matching for Network Applications. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[23] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A Matlab-like Environment for Machine Learning. In *Proceedings of the NIPS Workshop BigLearn*, 2011.

[24] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2016.

[25] D. J. Bernstein. A state-of-the-art message-authentication code. https://cr.yp.to/mac.html.

[26] D. J. Bernstein. The ChaCha family of stream ciphers. https://cr.yp.to/chacha.html.

[27] B. Dipak. Question about Fine grain and Coarse grain in OpenCL 2.0. https://community.amd.com/thread/169688.

[28] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2009.

[29] S. Han, K. Jang, K. Park, and S. B. Moon. PacketShader: a GPU-accelerated Software Router. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2010.

[30] M. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park. Kargus: a Highly-scalable Software-based Intrusion Detection System. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[31] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[32] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen. Raising the Bar for Using GPUs in Software Packet Processing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[33] J. Kim, S. Huh, K. Jang, K. Park, and S. Moon. The Power of Batching in the Click Modular Router. In *Proceedings of the Asia-Pacific Workshop on Systems (APSys)*, 2012.

[34] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon. NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2015.

[35] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, August 2000.

[36] V. Krasnov. It takes two to ChaCha (Poly). https://blog.cloudflare.com/it-takes-two-to-chacha-poly/.

[37] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[38] H. Liu and H. H. Huang. Enterprise: Breadth-First Graph Traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[39] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[40] O. Matz. Patchwork [dpdk-dev,v3,22/35] mempool: support no hugepage mode. http://www.dpdk.org/dev/patchwork/patch/12854/.

[41] Mellanox Technologies. Introduction to InfiniBand. http://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf.

[42] J. Park, W. Jung, G. Jo, I. Lee, and J. Lee. PIPSEA: A Practical IPsec Gateway on Embedded APUs. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[43] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the USENIX Large Installation System Administration Conference (LISA)*, 1999.

[44] SAPPHIRE Technology. AMD Merlin Falcon DB-FP4. http://www.sapphiretech.com/productdetial.asp?pid=F3F349C0-5835-407E-92C3-C19DEAB6B708&lang=eng.

[45] J. Tseng, R. Wang, J. Tsai, S. Edupuganti, A. W. Min, S. Woo, S. Junkins, and T.-Y. C. Tai. Exploiting Integrated GPUs for Network Packet Processing Workloads. In *Proceedings of the IEEE Conference on Network Softwarization (NetSoft)*, 2016.

[46] G. Vasiliadis and L. Koromilas. GASPP: A GPU-Accelerated Stateful Packet Processing Framework. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2014.

[47] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. MIDeA: A Multi-Parallel Intrusion Detection Architecture. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[48] W3Techs. Usage of SSL certificate authorities for websites. https://w3techs.com/technologies/overview/ssl_certificate/all.

Stateless Network Functions: Breaking the Tight Coupling of State and Processing

Murad Kablan, Azzam Alsudais, Eric Keller
University of Colorado, Boulder

Franck Le
IBM Research

Abstract

In this paper we present Stateless Network Functions, a new architecture for network functions virtualization, where we decouple the existing design of network functions into a stateless processing component along with a data store layer. In breaking the tight coupling, we enable a more elastic and resilient network function infrastructure. Our StatelessNF processing instances are architected around efficient pipelines utilizing DPDK for high performance network I/O, packaged as Docker containers for easy deployment, and a data store interface optimized based on the expected request patterns to efficiently access a RAMCloud-based data store. A network-wide orchestrator monitors the instances for load and failure, manages instances to scale and provide resilience, and leverages an OpenFlow-based network to direct traffic to instances. We implemented three example network functions (network address translator, firewall, and load balancer). Our evaluation shows (i) we are able to reach a throughput of 10Gbit/sec, with an added latency overhead of between 100μs and 500μs, (ii) we are able to have a failover which does not disrupt ongoing traffic, and (iii) when scaling out and scaling in we are able to match the ideal performance.

1 Introduction

As evidenced by their proliferation, middleboxes are an important component in today's network infrastructures [50]. Middleboxes provide network operators with an ability to deploy new network functionality as add-on components that can directly inspect, modify, and block or re-direct network traffic. This, in turn, can help increase the security and performance of the network.

While traditionally deployed as physical appliances, with Network Functions Virtualization (NFV), network functions such as firewalls, intrusion detection systems, network address translators, and load balancers no longer have to run on proprietary hardware, but can run in software, on commodity servers, in a virtualized environment, with high throughput [25]. This shift away from physical appliances should bring several benefits including the ability to elastically scale the network functions on demand and quickly recover from failures.

However, as others have reported, achieving those properties is not that simple [44, 45, 23, 49]. The central issue revolves around the state locked into the network functions – state such as connection information in a stateful firewall, substring matches in an intrusion detection system, address mappings in a network address translator, or server mappings in a stateful load balancer. Locking that state into a single instance limits the elasticity, resilience, and ability to handle other challenges such as asymmetric/multi-path routing and software updates.

To overcome this, there have been two lines of research, each focusing on one property[1]. For failure, recent works have proposed either (i) checkpointing the network function state regularly such that upon failure, the network function could be reconstructed [44], or (ii) logging all inputs (*i.e.,* packets) and using deterministic replay in order to rebuild the state upon failure [49]. These solutions offer resilience at the cost of either a substantial increase in per-packet latency (on the order of 10ms), or a large recovery time at failover (e.g., replaying all packets received since the last checkpoint), and neither solves the problem of elasticity. For elasticity, recent works have proposed modifying the network function software to enable the migration of state from one instance to another via an API [29, 45, 23]. State migration, however, takes time, inherently does not solve

[1] A third line, sacrifices the benefits of maintaining state in order to obtain elasticity and resilience [20].

the problem of unplanned failures, and as a central property relies on affinity of flow to instance – each rendering state migration a useful primitive, but limited in practice.

In this paper, we propose *stateless network functions* (or StatelessNF), a new architecture that breaks the tight coupling between the state that network functions need to maintain from the processing that network functions need to perform (illustrated in Figure 1). Doing so simplifies state management, and in turn addresses many of the challenges existing solutions face.

Resilience: With StatelessNF, we can instantaneously spawn a new instance upon failure, as the new instance will have access to all of the state needed. It can immediately handle traffic and it does not disrupt the network. Even more, because there is no penalty with failing over, we can failover much faster – in effect, we do not need to be certain a network function has failed, but instead only speculate that it has failed and later detect that we were wrong, or correct the problem (*e.g.,* reboot).

Elasticity: When scaling out, with StatelessNF, a new network function instance can be launched and traffic immediately directed to it. The network function instance will have access to the state needed through the data store (*e.g.,* a packet that is part of an already established connection that is directed to a new instance in a traditional, virtualized, firewall will be dropped because a lookup will fail, but with StatelessNF, the lookup will provide information about the established connection). Likewise, scaling in simply requires re-directing any traffic away from the instance to be shut down.

Asymmetric / Multi-path routing: In StatelessNF each instance will share all state, so correct operation is not reliant on affinity of traffic to instance. In fact, in our model, we assume any individual packet can be handled by any instance, resulting in an abstraction of a scalable, resilient, network function. As such, packets traversing different paths does not cause a problem.

While the decoupling of state from processing exists in other settings (*e.g.,* a web server with a backend database), the setting of processing network traffic, potentially requiring per packet updates to state, poses a significant challenge. A few key insights and advances have allowed us to bring this new design to a reality. First, there have been recent advances in disaggregated architectures, bringing with it new, low-latency and resilient data stores such as RAMCloud [39]. Second, not all state that is used in network functions needs to be stored in a resilient data store – only dynamic, network state needs to persist across failures and be available to all instances. State such as firewall rules, and intrusion detection system signatures can be replicated to each instance upon boot, as they are static state. Finally, network functions share a common pipeline design where there is typically a lookup operation when the packet is first being processed, and sometimes a write operation after the packet has been processed. This not only means there will be less interaction than one might initially assume, but also allows us to leverage this pattern to optimize the interactions between the data store and the network function instances to provide high performance.

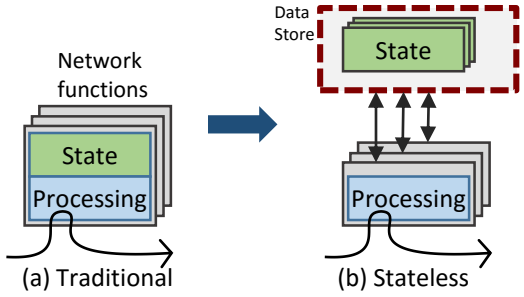


Figure 1: High level overview showing traditional network functions (a), where the state is coupled with the processing to form the network function, and stateless network functions (b), where the state is moved from the network function to a data store – the resulting network functions are now stateless.

We describe how four common network functions, can be re-designed in a stateless manner. We present the implementation of a stateful firewall, an intrusion prevention system, a network address translator, and a load balancer. Section 3 discusses the remote memory access. Section 6 discusses our utilization of RAMCloud for the data store, DPDK for the packet processing, and the optimized interface between the network functions and data store. Section 7 presents the evaluation: our experiments demonstrate that we are able to achieve throughput levels that are competitive with other software solutions [49, 45, 23] (4.6 Million packets per second for minimum sized packets), with only a modest penalty on per-packet latency (between 100us and 500us in the 95th percentile, depending on the application and traffic pattern). We further demonstrate the ability to seamlessly fail over, and scale out and scale in without any impact on the network traffic (as opposed to substantial disruption for a traditional design). Of course, the stateless network functions approach may not be suitable for *all* network functions, and there are further optimizations we can make to increase processing rates. This work, however, demonstrates that there is value for the functions we studied and that even with our current prototype, we are able to match processing rates of other systems with similar goals, while providing both scalability and resilience.

2 Motivation

Simply running virtual versions of the physical appliance counterparts provides operational cost efficiencies, but falls short in supporting the vision of a dynamic network infrastructure that elastically scales and is resilient to failure. Here, we illustrate the problems that the tight coupling of state and processing creates today, even with virtualized network functions, and discuss shortcomings of recent proposals.

First, we clarify our definition of the term "state" in this particular context. Although there is a variety of network functions, the state within them can be generally classified into (1) static state (*e.g.,* firewall rules, IPS signature database), and (2) dynamic state, which is continuously updated by the network function's processes [45, 21]. The latter can be further classified into (i) internal instance specific state (*e.g.,* file descriptors, temporary variables), and (ii) network state (*e.g.,* connection tracking state, NAT private to public port mappings). It is the dynamic network state that we are referring to that must persist across failures and be available to instances upon scaling in or out. The static state can be replicated to each instance upon boot, so will be accessed locally.

2.1 Dealing with Failure

For failure, we specifically mean crash (as opposed to byzantine) failures. Studies have shown that failures can happen frequently, and be highly disruptive [43].

The disruption comes mainly from two factors. To illustrate the first factor, consider Figure 2(a). In this scenario, we have a middlebox, say a NAT, which stores the mapping for two flows (F1 and F2). Upon failure, virtualization technology enables the quick launch of a new instance, and software-defined networking (SDN) [36, 14, 10] allows traffic to be redirected to the new instance. However, any packet belonging to flows F1 or F2 will then result in a failed lookup (no entry in the table exists). The NAT would instead create new mappings, which would ultimately not match what the server expects. This causes all existing connections to eventually timeout. Enterprises could employ hot-standby redundancy, but that doubles the cost of the network.

The second factor is due to the high cost of failover of existing solutions (further discussed below). As such, the mechanisms tend to be conservative when determining whether a device has failed [6] – if a device does not respond to one *hello* message, does that mean that the device is down, the network dropped a packet, or that the device is heavily loaded and taking longer to respond? Aggressive thresholds cause unnecessary failovers, resulting in downtime. Conservative thresholds may forward traffic to a device that has failed, resulting in disruption.

Problem with existing solutions

Two approaches to failure resilience have been proposed in the research community recently. First, pico replication [44] is a high availability framework that frequently checkpoints the state in a network function such that upon failure, a new instance can be launched and the state restored. To guarantee consistency, packets are only released once the state that they impact has been checkpointed – leading to substantial per-packet latencies (*e.g.,* 10ms for a system that checkpoints 1000 times per second, under the optimal conditions).

To reduce latency, another work proposes logging all inputs (*i.e.,* packets) coupled with a deterministic replay mechanism for failure recovery [49]. In this case, the per-packet latency is minimized (the time to log a single packet), but the recovery time is high (on the order of the time since last check point). In both cases, there is a substantial penalty – and neither deals with scalability or the asymmetric routing problem (discussed further in Section 2.3).

2.2 Scaling

As with the case of failover, the tight coupling of state and processing causes problems with scaling network functions. This is true even when the state is highly partitionable (*e.g.,* only used for a single flow of traffic, such as connection tracking in a firewall). In Figure 2(b), we show an example of scaling out. Although a new instance has been launched to handle the overloaded condition, existing flows cannot be redirected to the new instance – *e.g.,* if this is a NAT device, packets from flow F2 directed at the new instance will result in a failed lookup, as was the case with failure. Similarly, scaling in (decreasing instances) is a problem, as illustrated in Figure 2(c). As the load is low, one would like to shut down the instance which is currently handling flow F3. However, one has to wait until that instance is completely drained (*i.e.,* all of the flows it is handling complete). While possible, it is something that limits agility, requires special handling by the orchestration, and highly depends on flows being short lived.

Problem with existing solutions

The research community has proposed solutions based on state migration. The basic idea is to instrument the network functions with code that can export state from one instance and import that state into another instance. Router Grafting demonstrated this for routers (moving

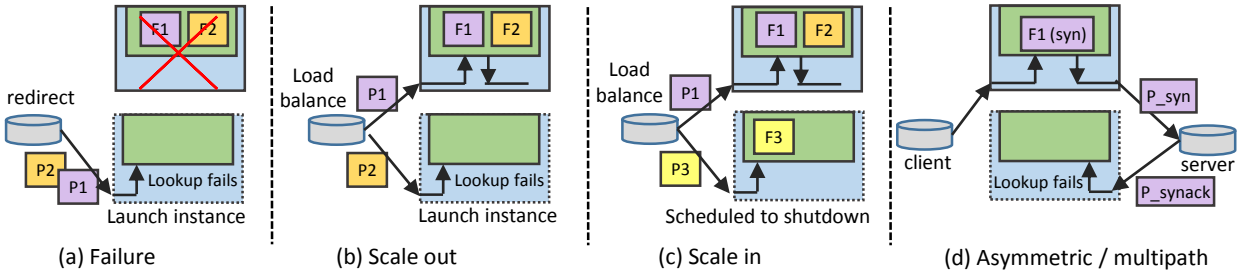


Figure 2: Motivational examples of traditional network functions and the problems that result from the tight coupling of state to the network function instance. State associated with some flow is labeled as F (*e.g.,* F2), and the associated packets within that flow are labeled as P (*e.g.,* P2).

BGP state) [29], and several have since demonstrated this for middleboxes [45, 23, 22] where partitionable state can be migrated between instances. State migration, however, takes time, inherently does not solve the problem of unplanned failures, and as a central property relies on affinity of flow to instance (limiting agility).

2.3 Asymmetric / Multi-path Routing

Asymmetric and multi-path routing can cause further challenges for a dynamic network function infrastructure: Asymmetric and multi-path [41] routing relates to the fact that traffic in a given flow may traverse different paths, and therefore be processed by different instances. For example, in the scenario of Figure 2(d), where a firewall has established state from an internal client connecting to a server (SYN packet), if the return syn-ack goes through a different firewall instance, this packet may result in a failed lookup and get dropped.

Problem with existing solutions

Recent work proposes a new algorithm for intrusion detection that can work across instances [35], but does so by synchronizing processing (directly exchanging state and waiting on other instances to complete processing as needed). Other solutions proposed in industry strive to synchronize state across middleboxes [31] (*e.g.,* HSRP [32]), but generally do not scale well.

3 How Network Functions Access State

The key idea in this paper is to decouple the processing from the state in network functions – placing the state in a data store. We call this stateless network functions (or StatelessNF), as the network functions themselves become stateless, and the statefulness of the applications (*e.g.,* a stateful firewall) is maintained by storing the state in a separate data store.

To understand the intuition as to why this is feasible, even at the rates network traffic needs to be processed, here we discuss examples of state that would be decoupled in common network functions, and what the access patterns are.

Table 1 shows the network state to be decoupled and stored in a remote storage for four network functions (TCP re-assembly is shown separate from IPS for clarity, but we would expect them to be integrated and reads/writes combined). As shown in the table, and discussed in Section 2, we only decouple network state.

We demonstrate how the decoupled state is accessed with pseudo-code of multiple network function algorithms, and summarize the needed reads and writes to the data store in Table 1. In all algorithms, we present updating or writing state to the data store as *writeRC* and reads as *readRC* (where RC relates to our chosen data store, RAMCloud). Below we describe Algorithms 1 (load balancer) and 2 (IPS). The pseudo-code of a stateful firewall, TCP re-assembly, and NAT are provided in Appendix for reference.

For the load balancer, upon receiving a TCP connection request, the network function retrieves the list of backend servers from the remote storage (line 4), and then assigns a server to the new flow (line 5). The load for the backend servers is subsequently updated (line 6), and the revised list of backend servers is written into remote storage (line 7). The assigned server for the flow is also stored into remote storage (line 8), before the packet is forwarded to the selected server. For a data packet, the network function retrieves the assigned server for that flow, and forwards the packet to the server.

Algorithm 2 presents the pseudo-code for a signature-based intrusion prevention system (IPS), which monitors network traffic, and compares packets against a database of signatures from known malicious threats using an algorithm such as Aho-Corasick algorithm [11] (as used

Network Function	State	Key	Value	Access Pattern
Load Balancer	Pool of Backend Servers	Cluster ID	IP List	1 read/write at start/end of conn.
	Assigned Server	5-Tuple	IP Address	1 read/write at start/end of conn. 1 read for every other packet
Firewall	Flow	5-Tuple	TCP Flag	5 read/write at start/end of conn. 1 read for every other packet
NAT	Pool of IPs and Ports	Cluster ID	IP and Port List	1 read/write at start/end of conn.
	Mapping	5-Tuple	(IP, Port)	1 read/write at start/end of conn. 1 read for every other packet
TCP Re-assembly	Expected Seq Record	5-Tuple	(Next Expected Seq, Keys for Buffered Pkts)	1 read/write for every packet
	Buffered Packets	Buffer Pointer	Packet	1 read/write for every out-of-order packet
IPS	Automata State	5-Tuple	Int	1 write for first packet of flow, 1 read/write for every other packet

Table 1: Network Function Decoupled States

Algorithm 1 Load Balancer

```
1:  procedure PROCESSPACKET(P: TCPPacket)
2:      extract 5-tuple from incoming packet
3:      if (P is a TCP SYN) then
4:          backendList ← readRC(Cluster ID)
5:          server ← nextServer(backendList, 5-tuple)
6:          updateLoad(backendList, server)
7:          writeRC(Cluster ID, backendList)
8:          writeRC(5-tuple, server)
9:          sendPacket(P, server)
10:     else
11:         server ← readRC(5-tuple)
12:         if (server is NULL) then
13:             dropPacket(P)
14:         else
15:             sendPacket(P, server)
```

Algorithm 2 IPS

```
1:  procedure PROCESSPACKET(P: TCPPacket)
2:      extract 5-tuple, and TCP sequence number from P
3:      if (P is a TCP SYN) then
4:          automataState ← initAutomataState()
5:          writeRC(5-tuple, automataState)
6:      else
7:          automataState ← readRC(5-tuple)
8:          while (b ← popNextByte(P.payload)) do
9:              // alert if found match
10:             // else, returns updated automata
11:             automataState ← process(b, automataState)
12:         writeRC(5-tuple, automataState)
13:         sendPacket(P)
```

in Snort [9]). At a high-level, a single deterministic automaton can be computed offline from the set of signatures (stored as static state in each instance). As packets arrive, scanning each character in the stream of bytes triggers one state transition in the deterministic automaton, and reaching an output state indicates the presence of a signature.

The 5-Tuple of the flow forms the key, and the state (to be stored remotely) simply consists of the state in the deterministic automaton (e.g., an integer value representing the node reached so far in the deterministic automaton). Upon receiving a new flow, the automata state is initialized (line 4). For a data packet, the state in the deterministic automaton for that flow is retrieved from remote storage (line 7). The bytes from the payload are then scanned (line 8). In the absence of a malicious signature, the updated state is written into remote storage (line 12), and the packet forwarded (line 13). Out-of-order packets are often considered a problem for Intrusion Prevention Systems [52]. Similar to the Snort TCP reassembly preprocessor [9], we rely on a TCP re-assembly module to deliver the bytes to the IPS in the proper order.

For the load balancer, we observe that we require one read for each data packet, and at most one additional read and write to the remote storage at the start and end of each connection. For the IPS, we observe that we require one write to the remote storage to initialize the automata state at the start of each connection, and one read and one write to remote storage for each subsequent data packet of the connection. Table 1 shows similar patterns for other network functions, and Section 7 analyzes the performance impact of such access patterns, and demonstrates that we can achieve multi Gbps rates.

4 Overall StatelessNF Architecture

At a high level, StatelessNF consists of a network-wide architecture where, for each network function application (*e.g.,* a firewall), we effectively have the abstraction of a single network function that reliably provides the necessary throughput at any given time. To achieve this, as illustrated in Figure 3, the StatelessNF architecture consists of three main components – the data store, the hosts to host the instances of the network function, and an orchestration component to handle the dynamics of the network function infrastructure. The network function hosts are simply commodity servers. We discuss the internal architecture of network function instances in Section 5. In this section, we elaborate on the data store and network function orchestration within StatelessNF.

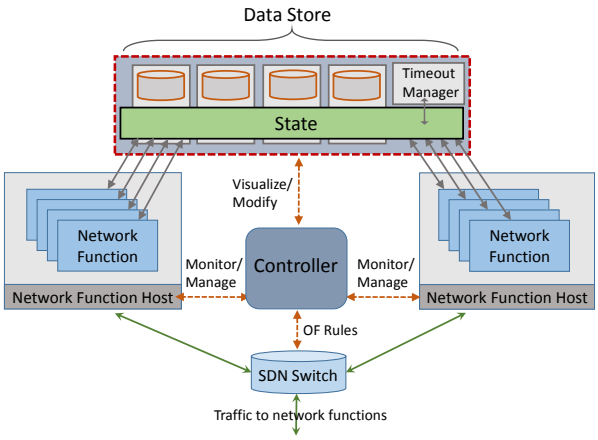


Figure 3: StatelessNF System Architecture

4.1 Resilient, Low-latency Data Store

A central idea in StatelessNF, as well as in other uses of remote data stores, is the concept of separation of concerns. That is, in separating the state and processing, each component can concentrate on a more specific functionality. In StatelessNF, a network function only needs to process network traffic, and does not need to worry about state replication, etc. A data store provides the resilience of state. Because of this separation, and because it resides on the critical path of packet processing, the data store must also provide low-latency access. For our purposes, we assume a data store that does not need support for transactions, but we anticipate exploring the impact of network functions that may require transactions as future work. In this paper, we choose RAMCloud [39] as our data store. RAMCloud is a distributed key-value storage system that provides low-latency access to data, and supports a high degree of scalability.

Resilient: For a resilient network function infrastructure, the data store needs to reliably store the data with high availability.

This property is common in available data stores (key value stores) through replication. For an in-memory data store, such as RAMCloud [39], the cost of replication would be high (uses a lot of RAM). Because of this, RAMCloud only stores a single copy of each object in DRAM, with redundant copies on secondary storage such as disk (on replica machines). To overcome the performance cost of full replication, RAMCloud uses a log approach where write requests are logged, and the log entry is what is sent to replicas, where the replicas fill an in-memory buffer, and then store on disk. To recover from a RAMCloud server crash, its memory contents must be reconstructed by replaying the log file.

Low-Latency: Each data store will differ, but RAMCloud in particular was designed with low-latency access in mind. RAMCloud is based primarily in DRAM and provides low-latency access (6μs reads, 15μs durable writes for 100 bytes data) at large-scale (*e.g.,* 10,000 servers). This is achieved both by leveraging low-latency networks (such as Infiniband and RDMA), being entirely in memory, and through optimized request handling. While Infiniband is not considered commodity, we believe it has growing acceptance (*e.g.,* Microsoft Azure provides options which include Infiniband [5]), and our architecture does not fundamentally rely on Infiniband – RAMCloud developers are working on other interfaces (*e.g.,* RoCE [47] and Ethernet with DPDK), which we will integrate and evaluate as they become available.

Going beyond a key-value store: The focus of data stores is traditionally the key-value interface. That is, clients can read values by providing a key (which returns the value), or write values by providing both the key and value. We leverage this key-value interface for much of the state in network functions.

The challenge in StatelessNF is that a common type of state in network functions, namely timers, do not effectively conform to a key-value interface. To implement with a key-value interface, we would need to continuously poll the data store – an inefficient solution. Instead, we extend the data store interface to allow for the creation and update of timers. The timer alert notifies one, and only one, network function instance, for which the handler on that instance processes the timer expiration.

We believe there may be further opportunities to optimize StatelessNF through customization of the data store. While our focus in this paper is more on the

network-wide capabilities, and single instance design, as a future direction, we intend to further understand how a data store can be adapted to further suit the needs of network functions.

4.2 Network Function Orchestration

The basic needs for orchestration involve monitoring the network function instances for load and failure, and adjusting the number of instances accordingly.

Resource Monitoring and Failure Detection: A key property of orchestration is being able to maintain the abstraction of a single, reliable, network function which can handle infinite load, but under the hood maintain as efficient of an infrastructure as possible. This means that the StatelessNF orchestration must monitor resource usage as well as be able to detect failure, and adjust accordingly – *i.e.,* launch or kill instances.

StatelessNF is not tied to a single solution, but instead we leverage existing monitoring solutions to monitor the health of network functions to detect failure as well as traffic and resource overload conditions. Each system hosting network functions can provide its own solution – *e.g.,* Docker monitoring, VMWare vcenter health status monitoring, IBM Systems Director for server and storage monitoring. Since we are using Docker containers as a method to deploy our network functions, our system consists of an interface that interacts with the Docker engines remotely to monitor, launch, and destroy the container-based network functions. In addition, our monitoring interface, through ssh calls, monitors the network function resources (cores, memory, and SR-IOV cards) to make sure they have enough capacity to launch and host network functions.

Important to note is that failure detection is different in StatelessNF than in traditional network function solutions. With StatelessNF, we have an effectively zero-cost to failing over – upon failure, any traffic that would go through the failed instance can be re-directed to any other instance. With this, we can significantly reduce the detection time, and speculatively failover. This is in contrast to traditional solutions that rely on timeouts to ensure the device is indeed failed.

Programmable Network: StatelessNF's orchestration relies on the ability to manage traffic. That is, when a new instance is launched, traffic should be directed to the instance; and when a failure occurs or when we are scaling-in, traffic should be redirected to a different instance. With emerging programmable networks, or software-defined networks (SDN), such as OpenFlow [36] and P4 [14], we can achieve this. Further, as existing SDN controllers (*e.g.,* ONOS [13], Floodlight [4], OpenDaylight [10]) provide REST APIs, we can integrate the control into our overall orchestration.

5 StatelessNF Instance Architecture

Whereas the StatelessNF overall architecture provides the ability to manage a collection of instances, providing the elasticity and resilience benefits of StatelessNF, the architecture of the StatelessNF instances are architected to achieve the deployability and performance needed. As shown in Figure 4, the StatelessNF instance architecture consists of three main components – (i) a packet processing pipeline that can be deployed on demand, (ii) high-performance network I/O, and (iii) an efficient interface to the data store. In this section, we elaborate on each of these.

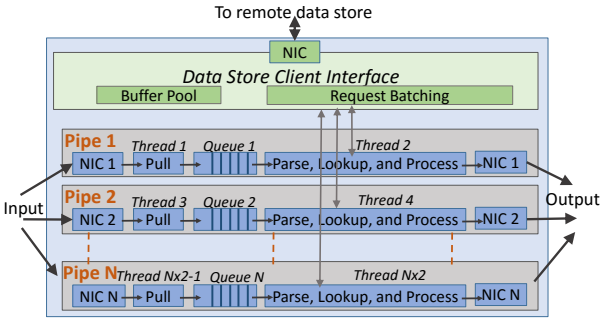


Figure 4: Stateless Network Function Architecture

5.1 Deployable Packet Processing Pipeline

To increase the performance and deployability of stateless network function instances, each network function is structured with a number of packet processing pipes. The number of pipes can be adaptive based on the traffic load, thus enabling a network function with a better resource utilization. Each pipe consists of two threads and a single lockless queue. The first thread is responsible for polling the network interface for packets and storing them in the queue. The second thread performs the main processing by dequeuing the packet, performing a lookup by calling the remote state interface to read, applying packet processing based on returned state and network function logic, updating state in the data store, and outputting the resulting packet(s) (if any).

Network function instances can be deployed and hosted with a variety of approaches – virtual machines, containers, or even as physical boxes. We focus on containers as our central deployable unit. This is due to

their fast deployment, low performance overhead, and high reusability. Each network function instance is implemented as a single process Docker instance with independent cores and memory space/region. In doing so, we ensure that network functions don't affect each other.

For network connectivity, we need to share the physical interface among each of the containers (pipelines). For this, we use SR-IOV[7] to provide virtual interfaces to each network function instance. Modern network cards have hardware support for classifying traffic and presenting to the system as multiple devices – each of the virtual devices can then be assigned to a network function instance. For example, our system uses Intel x520 server adapters[27] that can provide up to 126 virtual cards with each capable of reaching maximum traffic rate (individually). For connectivity to the data store, as our implementation focuses on RAMCloud, each network function host is equipped with a single Infiniband card that is built on the Mellanox RDMA library package[37], which allows the Infiniband NIC to be accessed directly from multiple network function user-space applications (bypassing the kernel). As new interfaces for RAMCloud are released, we can simply leverage them.

5.2 High-performance Network I/O

As with any software-based network processing application, we need high performance I/O in order to meet the packet processing rates that are expected. For this, we leverage the recent series of work to provide this – *e.g.,* through zero copy techniques. We specifically structured our network functions on top of the Data Plane Development Kit (DPDK) architecture [26]. DPDK provides a simple, complete framework for fast packet processing.

One challenge that arises with the use of DPDK in the context of containers is that large page support is required for the memory pool allocation used for packet buffers and that multiple packet processing pipes (containers) may run simultaneously on a single server. In our case, each pipe is assigned a unique page filename and specified socket memory amount to ensure isolation[2]. We used the DPDK Environment Abstraction Layer (EAL) interface for system memory allocation/de-allocation and core affinity/assignment procedures among the network functions.

[2] After several tests, we settled on 2GB socket memory for best performance.

5.3 Optimized Data Store Client Interface

Perhaps the most important addition in StatelessNF is the data store client interface. The importance stems from the fact that it is through this interface, and out to a remote data store, that lookups in packet processing occur. That is, it sits in the critical path of processing a packet and is the main difference between stateless network functions and traditional network functions.

Each data store will come with an API to read and write data. In the case of RAMCloud, for example, it is a key-value interface which performs requests via an RPC interface, and that leverages Infiniband (currently). RAMCloud also provides a client interface which abstracts away the Infiniband interfacing.

To optimize this interface to match the common structure of network processing, we make use of three common techniques:

Batching: In RAMCloud, a single read/write has low-latency, but each request has overhead. When packets are arriving at a high rate, we can aggregate multiple requests into a single request. For example, in RAMCloud, a single read takes 6μs, whereas a multi-read of 100 objects takes only 51μs (or, effectively 0.51μs per request). The balance here, for StatelessNF, is that if the the batch size is too small, we may be losing opportunity for efficiency gains, and too long (even with a timeout), we can induce higher latency than necessary waiting for more packets. Currently, we have a fixed batch size to match our experimental setup (100 objects), but we ultimately envision an adaptive scheme which increases or decreases the batch size based on the current traffic rates.

Pre-allocating a pool of buffers: When submitting requests to the data store, the client must allocate memory for the request (create a new RPC request). As this interface is in the critical path, we reduce the overhead for allocating memory by having the client reuse a preallocated pool of object buffers.

Eliminating a copy: When the data from a read request is returned from the data store to the client interface, that data needs to be passed to the packet processing pipeline. To increase the efficiency, we eliminate a copy of the data by providing a pointer to the buffer to the pipeline which issued the read request.

6 Implementation

The StatelessNF orchestration controller is implemented in Java with an admin API that realizes the implementation of elastic policies in order to determine when to

create or destroy network functions. At present, the policies are trivial to handle the minimal needs of handling failure and elasticity, simply to allow us to demonstrate the feasibility of the StatelessNF concept (see Section 7 for elasticity and failure experiments). The controller interacts with the Floodlight [4] SDN controller to steer the flows of traffic to the correct network function instances by inserting the appropriate OpenFlow rules. The controller keeps track of all the hosts and their resources, and network function instances deployed on top of them. Finally, the controller provides an interface to access and monitor the state in the data store, allowing the operator to have a global view of the network status.

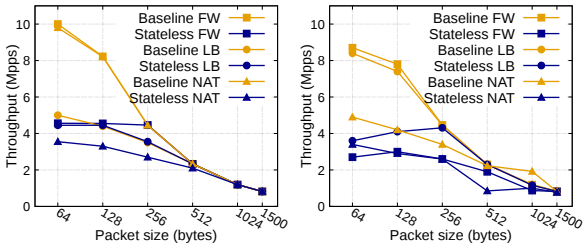
We implemented three network functions (firewall, NAT, load balancer) as DPDK [26] applications, and packaged as a Docker container. For each, we implemented in a traditional (non-stateless) and stateless fashion. In each case, the only difference is that the non-stateless version will access its state locally while the stateless version from the remote data store. The client interface to the data store is implemented in C++ and carries retrieval operations to RAMCloud [39]. The packet processing pipes are implemented in C/C++ in a sequence of pipeline functions that packets travel through, and only requires developers to write the application-specific logic – thus, making modifying the code and adding new network function relatively simple. The data store client interface and the packet processing pipes are linked at compile time.

7 Evaluation

This section evaluates the network functions performance, the recovery times in failure events, and the performance impact when scaling in/out with the proposed stateless architecture.

7.1 Experimental Setup

Our experimental setup is similar to the one depicted in Figure 3. It consists of six servers and two switches. Two servers are dedicated to hosting the network function instances. These two servers are connected via Infiniband to two other servers hosting RAMCloud (one acting as the RAMCloud coordinator, and the other server storing state), and are connected via Ethernet to a server acting as the traffic generator and sink (not shown in Figure 3). The last server hosts the StatelessNF controller which orchestrates the entire management. Specifically, we use the following equipment:



(a) Long flow case (b) Short flow case

Figure 5: Throughput of different packet sizes for long (a) and short (b) flows (*i.e.,* flow sizes >1000 and <100, respectively) measured in the number of packets per second.

- Network Function hosts: 2 Dell R630 Servers [16]: each has 32GB RAM, 12 cores (2.4GHz), one Intel 10G Server Adapter with SR-IOV support [27], and one 10G Mellanox InfiniBand Adapter Card [37].
- RAMCloud: 2 Dell R720 Servers [17], each with 48GB RAM, 12 cores (2.0GHz), one Intel 10G Server Adapter [27], one 10G Mellanox InfiniBand Adapter Card [37].
- Traffic generator/sink: 1 Dell R520 Servers [15]: 4GB RAM, 4 cores (2.0GHz), 2 Intel 10G Server Adapters [27] .
- Control: 1 Dell R520 Servers [15]: 4GB RAM, 4 cores (2.0GHz) to run StatelessNF and Floodlight controllers.
- SDN Switch: OpenFlow-enabled 10GbE Edge-Core [19].
- Infiniband Switch: 10Gbit Mellanox Infiniband switch between RAMCloud nodes and the network function hosts [38].

7.2 StatelessNF Performance

7.2.1 Impact of needing remote reads/writes

It is first critical to understand the performance of the RAMCloud servers as they may be a performance bottleneck, and limit the rates we can attain. Our benchmark tests reveal that a single server in RAMCloud can handle up to 4.7 Million lookup/sec. For write operations, a single server can handle up to 0.7 Million write/second.

The performance of a network function therefore heavily depends on the packets sizes, the network function's access patterns to the remote storage, and the processed traffic characteristics: For example, while a load

balancer requires three write operations per flow, a firewall requires five write operations per flow. As such, whether traffic consists of short flows (e.g., consisting of only hundreds of packets), or long flows (e.g., comprising tens of thousands of packets), these differences in access patterns can have a significant impact on the network function performance. In particular, short flows require many more writes for the same amount of traffic. We consequently distinguish three cases for the processed traffic: long, short, and average in regards to the size and number of flows. The long case consists of a trace of 3,000 large TCP flows of 10K packets each. The short case consists of a trace of 100,000 TCP flows of 100 packets each. Finally for the average case, we replayed a real captured enterprise trace [3] with 17,000 flows that range in size from 10 to 1,000 packets. In each case, we also varied the packet sizes to understand their impact on the performance. We used Tcpreplay with Netmap [51] to stream the three types of traces.

Figure 5 shows the throughput of StatelessNF middleboxes compared to their non-stateless counterparts (which we refer to as baseline) with long and short flows of different packet sizes. For minimum sized packets, we obtain throughputs of 4.6Mpps. For small sized packets (less than 128 bytes), the gap between stateless and non-stateless in throughput is due to a single RAMCloud server being able to handle around 4.7 Million lookups/sec. In contrast, in the baseline, all read and write operations are local. We highlight that such sizes are used to test the upper bound limits of our system.

As packets get larger in size (greater than 128 bytes), the rates of stateless and baseline network functions converge. The obtained throughputs are competitive with those of existing elastic and fail resilience software solutions [49, 45, 23]. To understand the performance of stateless network functions with real traces, we increase the rate of the real trace to more than the original rate at which it was captured[3], and analyze the achievable throughput. Since the packet sizes vary considerably (80 to 1500 bytes), we report the throughput in terms of traffic rate (Gbit/sec) rather than packets/sec. Figure 6 shows that the statelessNF firewall and loadbalancer have comparable performance than their baseline counterpart. The stateless NAT reaches a limit that is 1Gbps lower than the non-stateless version. Finally, we also observe that the performance of the NAT are several Gbps lower than the firewall and load balancer. This is due to the overhead of IP header checksum after modifying the packet IP addresses and port numbers.

[3]The rates of enterprise traces we found vary from 0.1 to 1 Gbit/sec

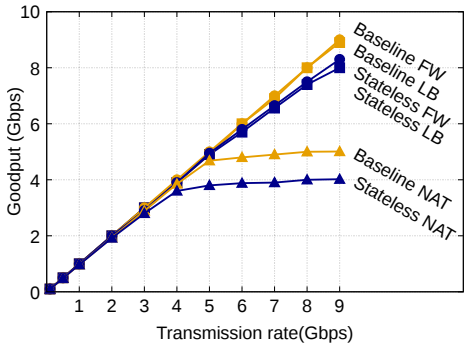
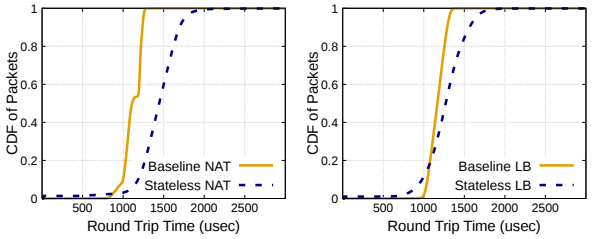


Figure 6: Measured goodput (Gbps) for enterprise traces.



(a) NAT (b) Load Balancer

Figure 7: Round-trip time (RTT) of packets.

7.2.2 Latency

The interaction with the remote storage can increase the latency of each packet, as every incoming packet must be buffered until its lookup operation is completed. To evaluate the delay increase, we compared the round-trip time (RTT) of each packet in the stateless and baseline network functions. We timestamp packets, send the traffic through the network function which resends the packets back to the initial host.

Figure 7 shows the cumulative distribution function (CDF) for the RTT of packets traversing the NAT and load balancer[4]. In the 50th percentile, the RTT of StatelessNF packets is only 100μs larger than the baseline's for the load balancer and NAT, and in the 95th percentile the RTT is only 300μs larger. The added delay we see in StatelessNF is a combination of read misses (which can reach 100μs), preparing objects for read requests from RAMCloud, casting returned data, and the actual latency of the request. These numbers are in the range of other comparable systems (*e.g.,* the low-latency rollback recovery system exhibited about a 300μs higher latency than the baseline [49]). Further, while the reported latency when using Ethernet (with DPDK) to communi-

[4]The RTT for firewall (both stateless and baseline) showed similar trend to load balancer with a better average delay (67μs less).

cate with RAMCloud is higher than Infiniband (31.1μs, read 100B, and 77μs write 100B), it is still less than the average packet delays reported with StatelessNF system (65us, 100us, and 300us for firewall, load balancer, and NAT respectively). Given the actual network traversals of requests can occur in parallel as the other aspects of the request, we believe that the difference in latency between Ethernet with DPDK and Infiniband can be largely masked. We intend to validate as future work.

7.2.3 IPS Analysis

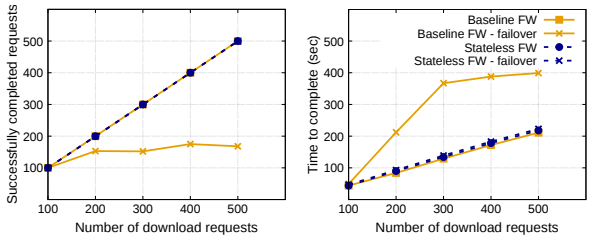
This section analyzes the impact of decoupling the state for an IPS. The overall processing of an IPS is more complex (Section 3) than the three network functions we previously analyzed. However, its access patterns to the remote storage is only incrementally more complex.

To analyze the impact of decoupling automaton state into remote storage, we implemented an in-line stateless network function that emulates a typical IPS in terms of accessing the automaton state and performs a read and write operation for every packet. For comparison, we run Snort [8] as an in-line IPS, and streamed real world enterprise traces through both instances: Snort and our stateless emulated IPS. The stateless emulated IPS was able to reach a throughput of 2.5Gbit/sec while the maximum performance for the Snort instance was only 2Gbit/sec. These results show that for an IPS, the performance bottleneck is the internal processing (*e.g.,* signature search), and not the read/write operations to the remote storage.

7.3 Failure

As we discussed in Section 3, in the case of failover, the instance we failover to can seamlessly handle the redirected traffic from the failed instance without causing any disruption for the traffic. To illustrate this effect, and compare to the traditional approach, we performed a number of file downloads that go through a firewall, and measured the number of successful file downloads and the time require to complete all of the downloads in the following cases: 1) baseline and stateless firewalls with no failure; 2) baseline and stateless firewall with failure where we redirect traffic to an alternate instance. In this case, we are only measuring the effect of the disruption of failover, as we assume a perfect failure detection, and simulate this by programming the SDN switch to redirect all traffic at some specific time. If we instrumented failure detection, the results would be more pronounced.

Figure 8 shows our results where we downloaded up to 500 20MB files in a loop of 100 concurrent http downloads through the firewall. As we can see, the baseline



(a) Completed requests (b) Time to complete requests

Figure 8: (a) shows the total number of successfully completed requests, and (b) shows the time taken to satisfy completed requests.

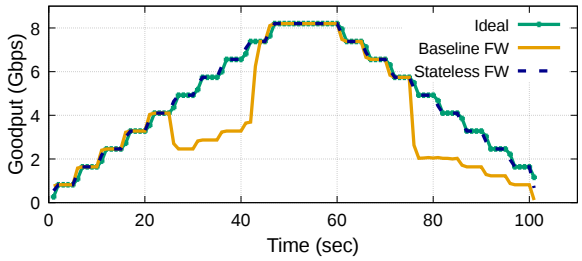


Figure 9: Goodput (Gbps) for stateless and baseline firewalls while scaling out (t=25s) and in (t=75s).

firewall is significantly affected by the sudden failure because the backup instance will not recognize the redirected traffic, hence will drop the connections, which in turn results in the client re-initiating the connections after a TCP connection timeout[5]. Not only was the stateless firewall able to successfully complete all downloads, but the performance was unaffected due to failure, and matched the download time of the baseline firewall when it did not experience failure.

7.4 Elasticity

In this paper, we claim that decoupling state from processing in network functions provides elasticity, where scaling in/out can be done with no disruption to the traffic. To evaluate StatelessNF's capability of scaling in and out, we performed the following experiment: we streamed continuous traffic of tcp packets while gradually increasing the traffic rate every 5 seconds (as shown in Figure 9), keep it steady for 5 seconds, and then start decreasing the traffic rate every 5 seconds. The three lines in Figure 9 represent: the ideal throughput (Ideal) which matches the send rate, the baseline firewall, and

[5]We significantly reduced the TCP connection timeout in Linux to 20 seconds, from the default of 7200 seconds.

the stateless firewall. The experiment starts with all traffic going through a single firewall. After 25 seconds, when the traffic transmitted reaches 4Gbit/sec, we split it in half and redirect it to a second firewall instance. Then after 25 seconds of decreasing the sending rate, we merge the traffic back to the first firewall instance.

As Figure 9 shows, the stateless firewall matches the base goodput. That is because the newly added firewall already has the state it needs to process the redirected packets, and therefore does not get affected by traffic redirection. On the other hand, with the baseline firewall, once the traffic is split, the second firewall starts dropping packets because it does not recognize them (*i.e.,* doesn't have state for those flows). Similarly, upon scaling in, the firewall instance does not have the state needed for the merged traffic and thus breaks the connections.

8 Discussion

The performance of our current prototype is not a fundamental limit of our approach. Here we discuss two aspects which can further enhance performance.

Reducing interactions with a remote data store: Fundamentally, if we can even further reduce the interactions with a remote data store, we can improve performance. Some steps in this direction that we intend to pursue as future work include: (i) reducing the penalty of read misses by integrating a set membership structure (*e.g.,* a bloom filter [2]) into the RAMCloud system so that we do not have to do a read if the data is not there, (ii) explore the use of caching for certain types of state (read mostly), and (iii) exploring placement of data store instances, perhaps even co-located with network function instances, in order to maintain the decoupled architecture, but allowing more operations to be serviced by the local instance and avoiding the consistency issues with cache (remote reads will still occur when the data isn't local, providing the persistent and global access to state).

Date store scalability We acknowledge that we will ultimately be limited by the scalability of the data store, but generally view data stores as scalable and an active area of research. In addition, while we chose RAMCloud for its low latency and resiliency, other systems such as FaRM [18] (from Microsoft) and a commercially available data store from Algo-Logic [1] report better throughput and lower latency, so we would see an immediate improvement if they become freely available.

9 Related Work

Beyond the most directly related work in Section 2, here we expand along three additional categories.

Disaggregation: The concept of decoupling processing from state follows a line of research in disaggregated architectures. [34], [33], and [40] all make the case for disaggregating memory into a pool of RAM. [24] explores the network requirements for an entirely disaggregated datacenter. In the case of StatelessNF, we demonstrate a disaggregated architecture suitable for the extreme use case of packet processing. Finally, this paper significantly expands on our previous workshop paper [28] with a complete and optimized implementation of the entire system and three network functions, and a complete evaluation demonstrating scalability and resilience.

Data plane processing: In addition to DPDK, frameworks like netmap [46] and Click [30] (particularly Click integrated with netmap and DPDK [12]) also provide efficient software packet processing frameworks, and therefore might be suitable for StatelessNF.

Micro network functions: The consolidated middlebox [48] work observed that course grained network functions often duplicate functionality as other network functions (*e.g.,* parsing http messages), and proposed to consolidate multiple network functions into a single device. In addition, e2 [42] provides a coherent system for managing network functions while enabling developers to focus on implementing new network functions. Each are re-thinking the architecture and complementary.

10 Conclusions and Future Work

In this paper, we presented stateless network functions, a novel design and architecture for network functions where we break the tight coupling of state and processing in network functions in order to achieve greater elasticity and failure resiliency. Our evaluation with a complete implementation demonstrates these capabilities, as well as demonstrates that we are able to process millions of packets per second, with only a few hundred microsecond added latency per packet. We do imagine there are further ways to optimize the performance and a desire for more network functions, and we leave that as future work. We instead focused on demonstrating the viability of a novel architecture which, we believe, fundamentally gets at the root of the important problem.

References

[1] Algo-logic systems. http://algo-logic.com/.

[2] Bloom filter. https://en.wikipedia.org/wiki/Bloom_filter.

[3] Digital corpora. http://digitalcorpora.org/corp/nps/scenarios/2009-m57-patents/net/.

[4] Floodlight. http://floodlight.openflowhub.org/.

[5] Microsoft Azure Virtual Machines. https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-linux-a8-a9\-a10-a11-specs/.

[6] Palo Alto Networks: HA Concepts. https://www.paloaltonetworks.com/documentation/70/pan-os/pan-os/high-availability/ha-concepts.html.

[7] Single-root IOV. https://en.wikipedia.org/wiki/Single-root_IOV.

[8] Snort IDS. https://www.snort.org.

[9] Snort Users Manual 2.9.8.3. http://manual-snort-org.s3-website-us-east-1.amazonaws.com/.

[10] The OpenDaylight Platform. https://www.opendaylight.org/.

[11] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 1975.

[12] T. Barbette, C. Soldani, and L. Mathy. Fast userspace packet processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 5–16, Washington, DC, USA, 2015. IEEE Computer Society.

[13] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pages 1–6. ACM, Aug 2014.

[14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[15] Dell. Poweredge r520 rack server. http://www.dell.com/us/business/p/poweredge-r520/pd.

[16] Dell. Poweredge r630 rack server. http://www.dell.com/us/business/p/poweredge-r630/pd.

[17] Dell. Poweredge r720 rack server. http://www.dell.com/us/business/p/poweredge-7520/pd.

[18] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414. USENIX Association, Apr 2014.

[19] Edge-Core. 10gbe data center switch. http://www.edge-core.com/ProdDtl.asp?sno=436&AS5610-52X.

[20] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2016.

[21] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Toward software-defined middlebox networking. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 7–12, New York, NY, USA, 2012. ACM.

[22] A. Gember-Jacobson and A. Akella. Improving the safety, scalability, and efficiency of network function state transfers. In *Proc. 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, Aug 2015.

[23] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella.

OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the ACM SIGCOMM*. ACM, Aug 2014.

[24] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *Proceedings of the 12th ACM Workshop on Hot Topics in Networks (HotNets)*. ACM, Nov 2013.

[25] J. Hwang, K. K. Ramakrishnan, and T. Wood. Netvm: High performance and flexible networking using virtualization on commodity platforms. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 445–458, Berkeley, CA, USA, 2014. USENIX Association.

[26] Intel. Data plane development kit. `http://dpdk.org`.

[27] Intel. Ethernet converged network adapter. `http://www.intel.com/content/www/us/en/ethernet-products/converged-network-adapters/ethernet-x520-server-adapters-brief.html`.

[28] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller. Stateless network functions. In *Proc. ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, 2015.

[29] E. Keller, J. Rexford, and J. Van Der Merwe. Seamless BGP Migration with Router Grafting. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 235–248. USENIX Association, Apr 2010.

[30] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, Aug. 2000.

[31] J. Kronlage. Stateful NAT with Asymmetric Routing. `http://brbccie.blogspot.com/2013/03/stateful-nat-with-asymmetric-routing.html`, March 2013.

[32] T. Li, B. Cole, P. Morton, and D. Li. RFC 2281: Cisco Hot Standby Router Protocol (HSRP), March 1998.

[33] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, Jun 2009.

[34] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *Proceedings of 18th IEEE International Symposium High Performance Computer Architecture (HPCA)*. IEEE, Feb 2012.

[35] J. Ma, F. Le, A. Russo, and J. Lobo. Detecting distributed signature-based intrusion: The case of multi-path routing attacks. In *IEEE INFOCOM*. IEEE, 2015.

[36] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, Aug 2008.

[37] Mellanox. Infiniband single/dual-port adapter. `http://www.mellanox.com/page/products_dyn?product_family=161&mtag=connectx_3_pro_vpi_card`.

[38] Mellanox. Infiniband switch. `http://www.mellanox.com/related-docs/prod_eth_switches/PB_SX1710.pdf`.

[39] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–41. ACM, Oct 2011.

[40] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 43(4), Jan. 2010.

[41] C. Paasch and O. Bonaventure. Multipath TCP. *Communications of the ACM*, 57(4):51–57, April 2014.

[42] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A Framework for NFV Applications. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Oct 2015.

[43] R. Potharaju and N. Jain. Demystifying the dark side of the middle: A field study of middlebox failures in datacenters. In *Proceedings of the 2013 Conference on Internet Measurement Conference (IMC)*, Oct 2013.

[44] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico replication: A high availability framework for middleboxes. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*. ACM, Oct 2013.

[45] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proceedings of the 10th USENIX Network System Design and Implementation (NSDI)*, pages 227–240. USENIX Association, April 2013.

[46] L. Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[47] RoCE. RDMA over Converged Ethernet. `https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet`.

[48] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 24–24, Berkeley, CA, USA, 2012. USENIX Association.

[49] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. a. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. Rollback-recovery for middleboxes. *SIGCOMM Comput. Commun. Rev.*, 45(4):227–240, Aug. 2015.

[50] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the ACM SIGCOMM*. ACM, Aug 2012.

[51] tcpreplay. Tcpreplay with netmap. `http://tcpreplay.appneta.com/wiki/howto.html`.

[52] X. Yu, W.-c. Feng, D. D. Yao, and M. Becchi. O3fa: A scalable finite automata-based pattern-matching engine for out-of-order deep packet inspection. In *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems (ANCS)*, Mar 2016.

A Appendix

Algorithm 3 TCP Re-assembly

```
1:  procedure PROCESSPACKET(P: TCPPacket)
2:      extract 5-tuple from incoming packet
3:      if (P is a TCP SYN) then
4:          record ← (getNextExpectedSeq(P), createEmpty-
    BufferPointerList())
5:          writeRC(5-tuple, record)
6:          sendPacket(P)
7:      else
8:          record ← readRC(5-tuple)
9:          if (record == NULL) then
10:             dropPacket(P);
11:         if (isNextExpectedSeq(P)) then
12:             record.expected ← getNextExpectedSeq(P)
13:             sendPacket(P)
14:             // check if we can send any packet in buffer
15:             while (bufferHasNextExpected-
    Seq(record.buffPtr, record.expected)) do
16:                 P ← readRC(pop(record.buffPtr).pktBuffKey)
17:                 record.expected ← getNextExpected-
    Seq(p)
18:                 sendPacket(P)
19:             writeRC(5-tuple, record)
20:         else
21:             // buffer packet
22:             pktBuffKey ← getPacketHash(P.header)
23:             writeRC(pktBuffKey, P)
24:             record.buffPtr ← insert(record.buffPtr, p.seq,
    pktBuffKey)
25:             writeRC(5-tuple, record)
```

Algorithm 4 Firewall

```
1:  procedure PROCESSPACKET(P: TCPPacket)
2:      key ← getDirectional5tuple(P, i)
3:      sessionState ← readRC(key)
4:      newState ← updateState(sessionState)
5:      if (stateChanged(newState, sessionState)) then
6:          writeRC(key, newState)
7:      if (rule-check-state(sessionState) == ALLOW) then
8:          sendPacket(P)
9:      else
10:         dropPacket(P)
```

Algorithm 5 NAT

```
1:  procedure PROCESSPACKET(P: Packet)
2:      extract 5-tuple from incoming packet
3:      (IP, port) ← readRC(5-tuple)
4:      if ((IP, Port) is NULL) then
5:          list-IPs-Ports ← readRC(Cluster ID)
6:          (IP, Port) ← select-IP-Port(list-IPs-Ports, 5-tuple)
7:          update(list-IPs-Ports, (IP, Port))
8:          writeRC(Cluster ID, list-IPs-Ports)
9:          writeRC(5-tuple, (IP, Port))
10:         extract reverse-5-tuple from incoming packet plus
    new IP-port
11:         writeRC(reverse-5-tuple, (P.IP, P.Port))
12:     P' ← updatePacketHeader(P, (IP, Port))
13:     sendPacket(P')
```

mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes

Muhammad Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park

School of Electrical Engineering, KAIST

Abstract

Stateful middleboxes, such as intrusion detection systems and application-level firewalls, have provided key functionalities in operating modern IP networks. However, designing an efficient middlebox is challenging due to the lack of networking stack abstraction for TCP flow processing. Thus, middlebox developers often write the complex flow management logic from scratch, which is not only prone to errors, but also wastes efforts for similar functionalities across applications.

This paper presents the design and implementation of mOS, a reusable networking stack for stateful flow processing in middlebox applications. Our API allows developers to focus on the core application logic instead of dealing with low-level packet/flow processing themselves. Under the hood, it implements an efficient event system that scales to monitoring millions of concurrent flow events. Our evaluation demonstrates that mOS enables modular development of stateful middleboxes, often significantly reducing development efforts represented by the source lines of code, while introducing little performance overhead in multi-10Gbps network environments.

1 Introduction

Network appliances or "middleboxes", such as intrusion detection systems and application accelerators, are widely deployed in modern networks [59]. With the trend towards commodity server-based middleboxes [59] and network functions virtualization [38], these middlebox applications are commonly implemented in software. Middlebox development, however, still remains an onerous task. It often requires handling complex flow-level states and events at layer 4 or above, such as connection state management and flow reassembly. The key challenge is that middlebox developers have to build these low-level flow management features from scratch, due to lack of common abstractions and well-defined APIs. This is in stark contrast to end-host applications programming, where application programmers rely on a set of networking system calls, such as the Berkeley socket API, that hides the details.

Existing socket APIs focus on end-to-end semantics and transferring application (layer 7) data. Unfortunately, they are not flexible enough to monitor session state, packet loss or retransmission patterns at lower layers. In contrast, popular packet processing frameworks, such as Click [46], DPDK [4], PacketShader IOEngine [40], and netmap [57], provide useful features for packet-level I/O processing, but lack flow-level abstraction required for stateful middlebox applications. A huge semantic gap exists between the two commonly-used abstractions. Thus, the state-of-the art middlebox programming remains that each application implements low-level flow-processing features in addition to the application-specific logic. This practice prevents code reuse and makes it challenging to understand the details of implementation. For example, we find that two popular NIDS implementations, Snort and Suricata, are drastically different, although they expose similar flow management features [19, 58].

This work presents the design and implementation of mOS, a reusable networking stack and an API for modular development of flow-processing middlebox applications. The design of mOS is based upon two principles. First, the API should facilitate a clear separation between low-level packet/flow processing and application-specific logic. While tight integration of the two layers might benefit performance, it easily becomes a source of complexity and a maintenance nightmare. In contrast, a reusable middlebox networking stack allows developers to focus on core middlebox application logic. Second, the middlebox networking API should provide programming constructs that natively support user-definable *flow events* for custom middlebox operations. Most middlebox operations are triggered by a set of custom flow events—being able to express them via a well-defined API is the key to modular middlebox programming. For example, a middlebox application that detects malicious payload in retransmission should be able to easily express the condition for the event and provide a custom action as its event handler. Building middlebox applications as a synthesis of event processing significantly improves the code readability while hiding the details for tracking complex conditions.

mOS satisfies a number of practical demands for middlebox development. First, it exposes a *monitoring socket* abstraction to precisely express the viewpoint of a middlebox on an individual TCP connection flowing through it. Unlike an end-host stack, mOS simultaneously manages the flow states of both end-hosts, which allows developers to compose arbitrary conditions of a flow state on either side. Second, mOS provides scalable monitoring. In high-speed networks with hundreds of thousands of concurrent flows, monitoring individual flow events incurs high overhead. Our event system significantly reduces the memory fooprint and memory bandwidth requirement for dynamic event registration and deregistration. Third, the mOS API supports fine-grained resource management on a per-flow basis. Developers can dynamically enable/disable event generation for an active flow and turn off tracking of unnecessary features. Tight controlling of computing resources leads to high performance as it avoids redundant cycle wastes. Finally, the mOS implementation extends the mTCP [43] codebase to benefit from the scalable user-level TCP stack architecture that harnesses modern multicore systems.

We make the following contributions. First, we present a design and implementation of a reusable flow-processing networking stack for modular development of high-performance middleboxes. Second, we present key abstractions that hide the internals of complex middlebox flow management. We find that the mOS monitoring socket and its flexible event composition provides an elegant separation between the low-level flow management and custom application logic. Third, we demonstrate its benefits in a number of real-world middlebox applications including Snort, Halfback, and Abacus.

2 Motivation and Approach

In this section, we explain the motivation for a unified middlebox networking stack, and present our approaches to its development.

2.1 Towards a Unified Middlebox Stack

mOS targets middleboxes that require L4-L7 processing but typically cannot benefit from existing socket APIs, including NIDS/NIPSes [3, 6, 19, 41, 58], L7 protocol analyzers [7, 9], and stateful NATs [5, 10]. These middleboxes track L4 flow states without terminating the TCP connections, often perform deep-packet inspection on flow-reassembled data, or detect anomalous behavior in TCP packet retransmission.

Unfortunately, developing flow management features for every new middlebox is very tedious and highly error-prone. As a result, one can find a long list of bugs related to flow management even in popular middleboxes [30–32, 34–36]. What is worse, some middleboxes fail to implement critical flow management functions. For example, PRADS [12] and nDPI [9] perform pattern matching, but they do not implement flow reassembly and would miss the patterns that span over multiple packets. Similarly, Snort's HTTP parsing module had long been packet-based [61] and vulnerable to pattern evasion attacks. While Snort has its own sophisticated flow management module, it is tightly coupled with other internal data structures, making it difficult to extend or to reuse.

A reusable middlebox networking stack would significantly improve the situation, but no existing packet processing frameworks meet the requirements of general-purpose flow management. `Click` [27, 37, 46] encourages modular programming of packet processing but its abstraction level is restricted to layer 3 or lower layers. `iptables` [5] along with its `conntrack` module supports TCP connection tracking, but its operation is based on individual packets instead of flows. For example, it does not support flow reassembly nor allows monitoring fine-grained TCP state change or packet retransmission. `libnids` [8] provides flow reassembly and monitors both server and client sides concurrently. Unfortunately, its reassembly logic is not mature enough to properly handle multiple holes in a receive buffer [17], and it does not provide any control knob to adjust the level of flow management service. Moreover, the performance of `iptables` and `libnids` depends on the internal kernel data structures that are known to be heavyweight and slow [43, 57]. `Bro` [55] provides flow management and events similar to our work, but its built-in events are often too coarse-grained to catch arbitrary conditions for middleboxes other than NIDS. While tailoring the event system to custom needs is possible through Bro's plugin framework [60], writing a plugin requires deep understanding of internal data structures and core stack implementation. In addition, `Bro` scripting and the implementation of its flow management are not designed for high performance, making it challenging to support multi-10G network environments.

2.2 Requirements and Approach

By analyzing various networking features of flow-processing middleboxes, we identify four key requirements for a reusable networking stack.

R1: *Middleboxes must be able to combine information from multiple layers using the stack.* For example, an NIDS must process each individual packet, but also be able to reconstruct bytestreams from TCP flows for precise analysis. A NAT translates the address of each IP packet, but it should also monitor the TCP connection setup and teardown to block unsolicited packets.

R2: *The networking stack must keep track of L4 states of both end-points, while adjusting to the level of state management service that applications require.* Tracking per-flow L4 state embodies multiple levels of services. Some applications require full TCP processing including

reassembling bi-directional bytestreams; others only need basic session management, such as sequence number tracking while a few require single-side monitoring without bytestream management (*e.g.* TCP/UDP port blockers). Thus, we must dynamically adapt to application needs.

R3: The networking stack must provide intuitive abstactions to cater middlebox developers' diverse needs in a modular manner. It must provide flow-level abstractions that enable developers to easily manipulate flows and packets that belong to the flow. It should enable separation of its services from the application logic and allow developers to create higher-level abstraction that goes beyond the basic framework. Unfortunately, current middlebox applications often do not decouple the two. For instance, Snort heavily relies on its customized `stream`[1] preprocessing module for TCP processing, but its application logic (the detection engine) is tightly interlaced with the module. A developer needs to understand the Snort-specific `stream` programming constructs before she can make any updates on flow management attributes inside the detection engine.

R4: *The networking stack must deliver high performance.* Many middleboxes require throughputs of 10+ Gbps handling hundreds of thousands of concurrent TCP flows even on commodity hardware [2, 41, 42, 44, 62].

Our main approach is to provide a well-defined set of APIs and a unified networking stack that hides the implementation details of TCP flow management from custom application logic. mOS consists of four components designed to meet all requirements:

- **mOS networking API** is designed to provide packet- and flow-level abstractions. It hides the details of all TCP-level processing, but exposes information from multiple layers **(R1, R2, and R3)**, encouraging developers to focus on the core logic and write a modular code that is easily reusable.
- **Unified flow management:** mOS delivers a composable and scalable flow management library **(R2 and R4)**. Users can adjust flow parameters (related to buffer management *etc.*) dynamically at run time. Moreover, it provides a multi-core aware, high performing stack.
- **User-defined events:** Applications built on mOS are flexible since the framework provides not only a set of built-in events but also offers support for registering user-defined events **(R1 and R3)**.
- **User-level stack:** mOS derives its high performance from mTCP [43] that is a parallelizable userspace TCP/IP stack **(R4)**.

3 mOS Programming Abstractions

In this section, we provide the design and the usage of the mOS networking API and explain how it simplifies stateful middlebox development.

[1]The `stream` module spans about 9,800 lines of code in Snort-3.0.0a1 version.

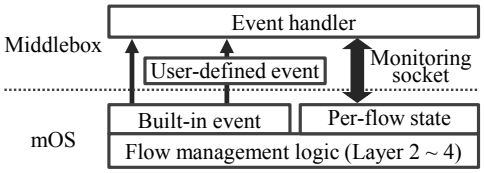


Figure 1: Interaction between mOS and its application

3.1 Monitoring Socket

The key abstraction that mOS exposes is a *monitoring socket*, which abstracts a middlebox's tap-point on a passing TCP flow or IP packets. Conceptually, it is similar to a Berkeley socket, but they differ in the operating semantics. First, a stream monitoring socket [2] represents a *non-terminating midpoint* of an active TCP connection. With a stream monitoring socket, developers write only high-level actions for an individual connection while underlying networking stack automatically tracks low-level TCP flow states of *both* client and server [3]. Second, a monitoring socket can monitor fine-grained TCP-layer operations while a Berkeley socket carries out coarse-grained, application-layer operations. For example, a monitoring socket can detect TCP or packet-level events such as abnormal packet retransmission, packet arrival order, abrupt connection termination, employment of weird TCP/IP options, etc., while it simultaneously supports reading flow-reassembled data from server or client.

Using the monitoring socket and its API functions (listed in Appendix A), one can write custom flow actions in a modular manner. First, a developer creates a 'passive' monitoring socket (similar to a listening socket) and binds it to a traffic scope, specified in a Berkeley packet filter (BPF) syntax. Only those flows/packets that fall into the scope are monitored. Note that there is no notion of "accepting" a connection since a middlebox does not engage in a connection as an explicit endpoint. Instead, one can express when custom action should be executed by setting up flow events as described in Section 3.2. All one needs is to provide the event handlers that perform a custom middlebox logic, since the networking stack automatically detects and raises the events by managing the flow contexts. When an event handler is invoked, it is passed an 'active' monitoring socket that represents the flow triggering the event. Through the socket, one can probe further on the flow state or retrieve and modify the last packet that raised the event. Figure 2 shows a code example that initializes a typical application with the mOS API.

3.2 Modular Programming with Events

The mOS API encourages modular middlebox programming by decomposing a complex application into a set of independent <event, event handler> pairs. It supports two classes of events: built-in and user-defined events.

[2]Similarly, a *raw* monitoring socket represents IP packets.

[3]We call a connection initiator as client and its receiver as server.

Event	Description
MOS_ON_PKT_IN	In-flow TCP packet arrival
MOS_ON_CONN_START	Connection initiation (the first SYN packet)
MOS_ON_REXMIT	TCP packet retransmission
MOS_ON_TCP_STATE_CHANGE	TCP state transition
MOS_ON_CONN_END	Connection termination
MOS_ON_CONN_NEW_DATA	Availability of new flow-reassembled data
MOS_ON_ORPHAN	Out-of-flow (or non-TCP) packet arrival
MOS_ON_ERROR	Error report (e.g., receive buffer full)

Table 1: mOS built-in events for stream monitoring sockets. Raw monitoring sockets can use only MOS_ON_PKT_IN raised for every incoming packet.

Built-in events represent pre-defined conditions of notable flow state that are automatically generated in the process of TCP flow management in mOS. Developers can create their own user-defined events (UDEs) by extending existing built-in or user-defined events.

Built-in event: Built-in events are used to monitor c mon L4 events in an active TCP connection, such start/termination of a connection, packet retransmissi or availability of new flow-reassembled data. With a st transition event, one can even detect any state change i TCP state transition diagram. Table 1 lists eight built events that we have drawn from common functionalities existing flow-processing middleboxes. These pre-defi events are useful in many applications that require ba flow state tracking. For example, developers can eas write a stateful NAT (or firewall) with only packet arri and connection start/teardown events without explic tracking sequence/acknowledgment numbers or TCP st transitions. Also, gathering flow-level statistics at a work vantage point can be trivially implemented in onl few lines of code as shown in Appendix B.

User-defined event: For many non-trivial middlebox plications, built-in events are insufficient since they of require analyzing L7 content or composing multiple c ditions into an event. For example, an event that detect duplicate ACKs or that monitors an HTTP request does not have built-in support.

UDEs allow developers to systematically express such desired conditions. A UDE is defined as a base event and a boolean filter function that specifies the event condition. When the base event for an UDE is raised, mOS fires the UDE only if the filter function is evaluated to true. mOS also supports a multi-event filter function that can dynamically determine an event type or raise multiple events simultaneously. This feature is useful when it has to determine the event type or trigger multiple related events based on the same input data.

UDEs bring three benefits to event-driven middlebox development. First, new types of events can be created in a flexible manner because the filter function can evaluate arbitrary conditions of interest. A good filter function, however, should run fast without producing unnecessary

```
1  static void
2  mOSAppInit(mctx_t m)
3  {
4    monitor_filter_t ft = {0};
5    int s; event_t hev;
6
7    // creates a passive monitoring socket with its scope
8    s = mtcp_socket(m, AF_INET, MOS_SOCK_MONITOR_STREAM, 0);
9    ft.stream_syn_filter = "dst net 216.58 and dst port 80";
10   mtcp_bind_monitor_filter(m, s, &ft);
11
12   // sets up an event handler for MOS_ON_REXMIT
13   mtcp_register_callback(m,s,MOS_ON_REXMIT,MOS_HK_RCV,OnRexmitPkt);
14
15   // defines a user-defined event that detects an HTTP request
16   hev = mtcp_define_event(MOS_ON_CONN_NEW_DATA, IsHTTPRequest, NULL);
17
18   // sets up an event handler for hev
19   mtcp_register_callback(m, s, hev, MOS_HK_RCV, OnHTTPRequest);
20 }
```

Figure 2: Initialization code of a typical mOS application. Due to space limit, we omit error handling in this paper.

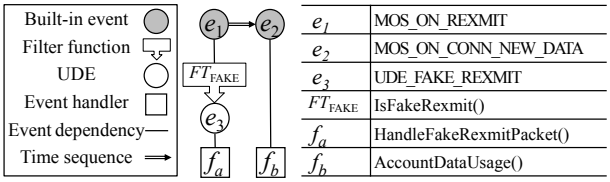


Figure 3: Abacus event-action diagram

side effect. Second, UDEs provide easy extensibility. One can create new events by extending any existing ones, including another UDE. For example, a developer can define a UDE that detects a YouTube [23] video request by extending a generic HTTP request UDE (e.g., using hev in Figure 2 as a base event). Third, it encourages code reuse. One can share a well-designed set of event definitions as a UDE library, and 3rd party developers can implement their own event handlers. For example, an open-source NIDS can declare all corner cases in flow management as a UDE library while 3rd party can provide custom actions to address each case.

3.3 Programming a Custom Middlebox

We show how one can build a custom flow-processing middlebox application using the mOS API and events. We pick Abacus [39] as an example here since it represents the needs of a real-world custom middlebox. Abacus is a cellular data accounting system that detects a "free-riding" attack by TCP-level tunneling. It has been reported that some cellular ISPs do not account for TCP retransmission packets [39], and this attack enables free-riding on cellular data by appending fake TCP headers that look like packet retransmission. The attack detection requires comparing the payload of original and retransmitted packets either by buffering a sliding window or sampling some bytes [39].

Writing Abacus with the mOS API is straightforward. First, we draw an event-action diagram that captures its main operations as shown in Figure 3. It represents the normal data as a built-in event (e_2, new data event) and registers a per-flow accounting function (f_b) for the event.

To detect the free-riding attack, we extend a built-in event (retransmission) to define a fake retransmission event (e_3). The filter function (FT_{FAKE}), as shown in Figure 4, determines whether the current packet retransmission is legal. In the code, `mtcp_getlastpkt()` retrieves the metadata (`pkt_info` structure) and the payload of the packet that triggers the retransmission event. `mctx_t` represents the thread context that the mOS stack is bound to and `sock` identifies the active flow that the packet belongs to. Then, the code uses `mtcp_ppeek()` to fetch the original flow-reassembled data at a specified sequence number offset and compares it with the payload if the sequence number ranges match. In case of partial retransmission, it calls `mtcp_getsockopt()` to retrieve non-contiguous TCP data fragments (`frags`) from a right flow buffer and compares the payload of the overlapping regions. If any part is different from the original content, it returns `true` and e_3 is triggered, or otherwise it returns `false`. When e_3 is triggered, f_a is executed to report an attack, and stops any subsequent event processing for the flow.

While Abacus is a conceptually simple middlebox, wri ing its flow management from scratch would require lot of programming effort. Depending on a middlebox, a least thousands to tens of thousands of code lines are re quired to implement basic flow management and variou corner cases. The mOS API and its stack significantly sav this effort while it allows the developer to write only th high-level actions in terms of events. Drawing an even action diagram corresponds well to the application desig process. Also, it better supports modular programmin since the developer only needs to define their own UDE and convert filters and event handlers into functions.

```
1  static bool
2  IsFakeRexmit(mctx_t mctx, int sock, int side, event_t event,
3               struct filter_arg *arg)
4  {
5    struct pkt_info pi;
6    char buf[MSS];
7    struct tcp_ring_fragment frags[MAX_FRAG_NUM];
8    int nfrags = MAX_FRAG_NUM;
9    int i, size, boff, poff;
10
11   // retrieve the current packet information
12   mtcp_getlastpkt(mctx, sock, side, &pi);
13
14   // for full retransmission, compare the entire payload
15   if (mtcp_ppeek(mctx, sock, side, buf,
16                  pi.payloadlen, pi.offset) == pi.payloadlen)
17     return memcmp(buf, pi.payload, pi.payloadlen);
18
19   // for partial retransmission, compare the overlapping region
20   // retrieve the data fragments and traverse them
21   mtcp_getsockopt(mctx, sock, SOL_MONSOCKET, (side == MOS_SIDE_CLI) ?
22           MOS_FRAGINFO_CLIBUF : MOS_FRAGINFO_SVRBUF, frags, &nfrags);
23
24   for (i = 0; i < nfrags; i++) {
25     if ((size = CalculateOverlapLen(&pi, &(frags[i]), &boff, &poff)))
26       if (memcmp(buf + boff, pi.payload + poff, size))
27         return true;    // payload mismatch detected
28   }
29   return false;
30 }
```

4 mOS Design and Implementation

In this section, we present the internals of mOS. At a high-level, mOS takes a stream of packets from a network, classifies and processes them by the flow, and triggers matching flow events. Inside event handlers, the application runs custom logic. mOS supports TCP flow state management for end-hosts, scalable event monitoring, extended flow reassembly, and fine-grained resource management. It is implemented by extending mTCP [43]. In total, it amounts to 27K lines of C code, which includes 11K lines of the mTCP code.

4.1 Stateful TCP Context Management

Automatic management of TCP contexts is the core functionality of mOS. For flow management, mOS keeps track of the following L4 states of both end-points: (1) TCP connection parameters for tracking initiation, state transition, and termination of each connection, (2) a payload reassembly buffer and a list of fragmented packets for detecting new payload arrival and packet retransmission. We further explain the payload reassembly buffer in Section 4.3.

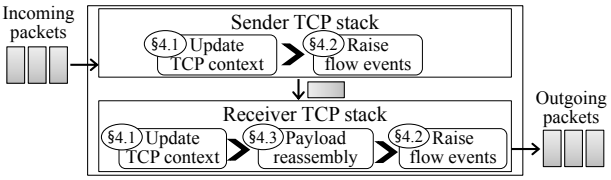
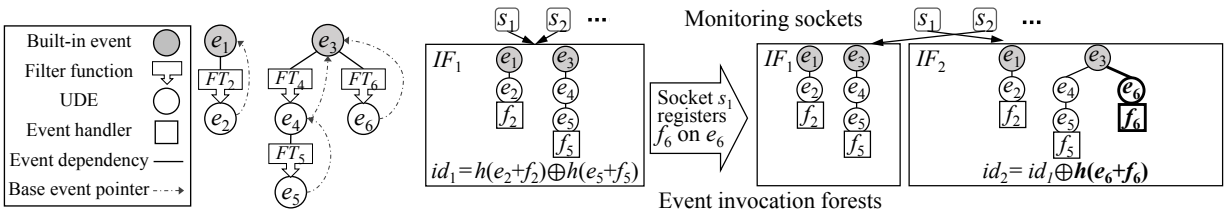


Figure 5: Packet processing steps in mOS

We design the TCP context update in mOS to closely reflect the real TCP state machine by emulating the states of both end-hosts. Tracking the flow states of both end-hosts is required as each side may take on a different TCP state. Figure 5 illustrates our model. When a packet arrives, mOS first updates its TCP context for the packet sender [4] and records all flow events that must be triggered. Note that event handlers are executed as a batch after the TCP context update. This is because intermixing them can produce an inconsistent state as some event handler may modify or drop the packet. Also, processing sender-side events before the receiver side's is necessary to strictly enforce the temporal order of the events. After sender-side stack update, mOS repeats the same process (update and trigger events) for the receiver-side stack. Any events relating to packet payload (new data or retransmission) are triggered in the context of a receiver since application-level data is read by the receiver. In addition, packet modification (or drop) is allowed only in the sender-side event handlers as mOS meticulously follows the middlebox semantics. The only exception is the retransmission event, which is processed just before receiver context update. This is to give

[4] Note that both server and client can be a packet sender.



(a) Global event dependency forest (d-forest) (b) Dynamic event registration upon event invocation forests (i-forests)

Figure 6: mOS's event management. s_1 and s_2 originally share the same event invocation forest (i-forest), IF_1. If s_1 registers an event handler, f_6, then IF_2 is created. If s_2 repeats the same registration later, then s_2 simply moves its i-forest pointer to IF_2.

its event handler a chance to modify or drop the packet if the retransmission turns out to be malicious. While a receiver-side event handler cannot modify the packet, it can still reset a connection in case it detects malicious intrusion attempts in reassembled payload.

4.2 Scalable Event Management

mOS applications operate by registering for flow events and providing custom actions for them. For high performance, it is critical to have scalable event management with respect to the number of flows and user-defined events. However, a naïve implementation introduces a serious scalability challenge because mOS allows events to be registered and de-registered on a per-flow basis for fine-grained control. A busy middlebox that handles 200K concurrent flows and 1,000 UDEs per flow [5] amounts to managing a billion events [6], which would require large amount of memory and consume huge memory bandwidth. mOS provides an efficient implementation of internal data structures and algorithms designed to address the scalability challenge.

Data structures: Note UDEs form a tree hierarchy with its root being one of the eight built-in events. Thus, mOS maintains all custom flow event definitions in a global event dependency forest (d-forest) with eight dependency trees (d-trees). Figure 6(a) shows an example of a d-forest. Each node in a d-tree represents a UDE as a base event (e.g., parent node) and its filter function. Separate from the d-forest, mOS maintains, for each monitoring socket, its event invocation forest (i-forest) that records a set of flow events to wait on. Similar to the d-forest, an i-forest consists of event invocation trees (i-trees) where each i-tree maintains the registered flow events derived from its root built-in event. Only those events with an event handler are being monitored for the socket.

Addressing scalability challenge: If each socket maintains a separate i-forest, it would take up a large memory footprint and cause performance degradation due to redundant memory copying and releasing of the same i-forest. mOS addresses the problem by *sharing the same i-forest* with different flows. Our observation is that flows of the same traffic class (e.g., Web traffic) are likely to process the same set of events, and it is highly unlikely for all sockets to have a completely different i-forest. This implies that we can reduce the memory footprint by sharing the same i-forest.

When an active socket (e.g., individual TCP connection) is created, it inherits the i-forest from its passive monitoring socket (e.g., listening socket) by keeping a pointer to it. When an event is registered or de-registered for an active socket, mOS first checks if the resulting i-forest already exists in the system. If it exists, the active socket simply shares the pointer to the i-forest. Otherwise, mOS creates a new i-forest and adds it to the system. In either case, the socket adjusts the reference count of both previous and new i-forests, and removes the i-forest whose reference count becomes zero.

The key challenge lies in how to efficiently figure out if the same i-forest already exists in the system. A naïve implementation would require traversing every event node in all i-forests, which does not scale. Instead, we present an O(1) solution here. First, we devise a novel i-forest identification scheme that produces a unique id given an i-forest. We represent the id of an i-forest with m i-trees as: $t_1 \oplus t_2 \oplus \ldots \oplus t_m$, where t_k indicates the id of the k-th i-tree and $\oplus$ is a bitwise exclusive-or (`xor`) operation. Likewise, the id of an i-tree with n leaf event nodes is defined as: $h(e_1+f_1) \oplus h(e_2+f_2) \oplus \ldots \oplus h(e_n+f_n)$, where h is a one-way hash function, $+$ is simple memory concatenation, and e_i and f_i are the ids of the i-th event and its event handler, respectively. Note that the ids of a distinct event and its event handler are generated as unique in the system, which makes a distinct i-forest have a unique id with a high probability with a proper choice of the hash function. We include the id of an event handler in hash calculation since some event can be registered multiple times with a different handler. Calculating the new id of an i-forest after adding or removing an event becomes trivial due to the `xor` operation. Simply, new-id $=$ old-id $\oplus$ $h(e+f)$ where e and f are the ids of the event and its event handler that need to be registered or deregistered. The fast id operation enables efficient lookup in the invocation forest hash table.

[5] Not unreasonable for an NIDS with thousands of attack signatures.

[6] 200K flows x 5,000 event nodes, assuming a UDE is derived from four ancestor base events on average.

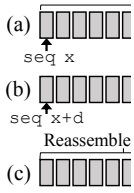


Figure 7: (a) W
ON_ERROR to no
nal read pointer
data fragments c

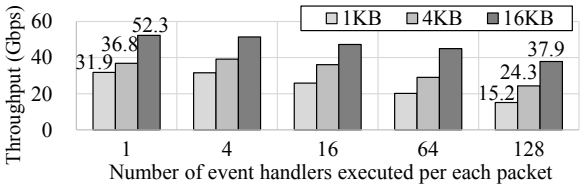


Figure 8: mOS performance over the number of events triggered per each packet. Performance measured with 192K concurrent connections fetching HTTP objects using a 60Gbps link. Event filters/handlers do minimal operation.

Deterministic
single packet a
Thus, the orde
defined for de
a fixed priorit
rival events (M
processed because they convey the L3 semantics. Then, `MOS_ON_CONN_START` is triggered followed by `MOS_ON_TCP_STATE_CHANGE` and `MOS_ON_CONN_NEW_DATA`. Finally, `MOS_ON_CONN_END` is scheduled to give a chance to other events to handle the flow data before connection termination. Note, all built-in events are handled after TCP context update with an exception of `MOS_ON_REXMIT`, a special event triggered just before receive-side TCP context update.

All derived events inherit the priority of their root built-in event. mOS first records all built-in events that are triggered, and 'executes' each invocation tree in a forest by the priority order of the root built-in events. 'Executing' an invocation tree means traversing the tree in the breadth-first search order and executing each node by evaluating its event filter and running its event handler. For example, events in F_2 in Figure 6(b) are traversed in the order of $e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4 \rightarrow e_6 \rightarrow e_5$.

4.3 Robust Payload Reassembly

Many middleboxes require L7 content scanning for detecting potential attack or application-specific patterns. mOS supports such applications with robust payload reassembly that handles a number of sophisticated cases.

Basic operation: mOS exposes `mtcp_peek()` and `mtcp_ppeek()` to the application for reading L7 data in a flow. Similar to `recv()`, `mtcp_peek()` allows the application to read the entire bytestream from an end-host. Internally, mOS maintains and adjusts a current read pointer for each flow as the application reads the data. `mtcp_ppeek()` is useful for retrieving flow data or fragments at an arbitrary sequence number.

Reassembly buffer outrun: Since TCP flow control applies between end-hosts, the receive buffer managed by mOS can become full while new packets continue to arrive (see Figure 7 (a)). Silent content overwriting is undesirable since the application may not notice the buffer overflow. Instead, mOS raises an error event to explicitly notify the application about the buffer overflow. The application can either drain the buffer by reading the data or enlarge the buffer. Otherwise, mOS overwrites the buffer with the new data, and adjusts the internal read pointer(see Figure 7 (b)). To notify the application about the overwriting, we make `mtcp_peek()` fail right after overwriting. Subsequent function calls continue to read the data from the new position. Notifying the application about buffer overflow and overwriting allows the developer to write correct operations even at corner cases.

Out-of-order packet arrival: Unlike the end-host TCP stack, some middlebox applications must read partially-assembled data, especially when detecting attack scenarios with out-of-order or retransmitted packets. mOS provides data fragment metadata by `mtcp_getsockopt()`, and the application can retrieve payload of data fragment by `mtcp_ppeek()` with a specific sequence number to read (see Figure 7 (c)).

Overlapping payload arrival: Another issue lies in how to handle a retransmitted packet whose payload overlaps with the previous content. mOS allows to express a flexible policy on content overlap. Given that the update policy differs by the end-host operating systems [53], mOS supports both policies (e.g., overwriting with the retransmitted payload or not) that can be configured on a per-flow basis. Or a developer can register for a retransmission event and implement any custom policy of her choice.

4.4 Fine-grained Resource Management

A middlebox must handle a large number of concurrent flows with limited resources. mOS is designed to adapt its resource consumption to the computing needs as follows.

Fine-grained control over reassembly: With many concurrent flows, the memory footprint and memory bandwidth consumption required for flow reassembly can be significant. This is detrimental to those applications that do not require flow reassembly. To support such applications, mOS allows disabling or resizing/limiting the TCP receive buffer at run-time on a per-flow basis. For example, middleboxes that rely on IP whitelisting modules (*e.g.* Snort's IP reputation preprocessor [18]) can use this feature to dynamically disable buffers for those flows that arrive from whitelisted IP regions.

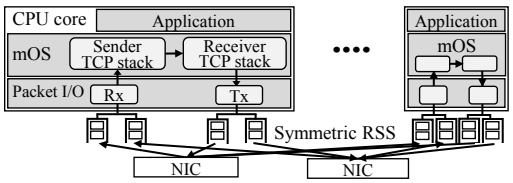


Figure 9: mOS application threading model

Uni-directional monitoring: Some middleboxes may want to monitor only the client-side requests or others deployed before server farms may be interested only in the ingress traffic. In such a case, developers can turn off TCP state management of one side. Disabling the TCP stack of one side would ignore raising events, stack update, and flow reassembly.

Dynamic event management: The number of registered events affects the overall performance, as shown in Figure 8. When a large number of UDEs are naïvely set up for all flows, it degrades the performance due to frequent filter invocations. To address this, the mOS API supports dynamic cancellation of registered events. For example, one can stop scanning the flow data for attack signatures beyond a certain byte limit [51]. Alternatively, one can register for a new event or switch the event filter depending on the characteristics of each flow. Such a selective application of flow events provides flexibility to the developers, while minimizing the overall resource consumption.

Threading model: mOS adopts the shared-nothing parallel-processing architecture that effectively harnesses modern multi-core CPUs. Figure 9 shows the threading model of mOS. At start, it spawns n independent threads, each of which is pinned to a CPU core and handles its share of TCP flows using symmetric receive-side scaling (S-RSS) [63]. S-RSS maps all packets in the same TCP connection to the same RX queue in a network interface card (NIC), by making the Toeplitz hash function [47] produce the same value even if the source and destination IP/port pairs on a packet are swapped. This enables line-rate delivery of packets to each thread as packet classification is done in NIC hardware. Also, it allows each mOS thread to handle entire packets in a connection without sharing flow contexts with other threads, which avoids expensive inter-core locks and cache interference. We adopt flow-based load balancing as it is reported to achieve a reasonably good performance with real traffic [63].

mOS reads multiple incoming packets as a batch but processes each packet by the run-to-completion model [28]. mOS currently supports Intel DPDK [4] and netmap [57] as scalable packet I/O, and supports the pcap library [20] for debugging and testing purposes. Unlike mTCP, the application runs event handlers in the same context of the mOS thread. This ensures fast event processing without context switching.

Appl	Modified	SLOC	Output
Snort	2,104	79,889	Stateful HTTP/TCP inspection
nDPI	765	25,483	Stateful session management
PRADS	615	10,848	Stateful session management
Abacus	-	4,639 → 561	Detect out-of-order packet tunneling

Table 2: Summary of mOS application updates. Snort's SLOC represents the code lines that are affected by our porting.

5 Evaluation

This section evaluates mOS by answering three key questions: (1) Does the mOS API support diverse use cases of middlebox applications? (2) Does mOS provide high performance? (3) Do mOS-based applications perform correct operations without introducing non-negligible overhead?

5.1 mOS API Evaluation

For over two years of mOS development, we have built a number of middlebox applications using the mOS API. These include simple applications such as a stateful NAT, middlebox-netstat, and a stateful firewall as well as porting real middlebox applications, such as Snort [58], nDPI library [9], and PRADS [12] to use our API. Using these case studies, we demonstrate that the mOS API supports diverse applications and enables modular development by allowing developers to focus on the core application logic. As shown in Table 2, it requires only 2%-12% of code modification to adapt to the mOS framework. Moreover, our porting experience shows that mOS applications have clear separation of the main logic from the flow management modules. We add a prefix 'm', to the name of mOS-ported application (*e.g.*, Snort → mSnort).

mSnort3: We demonstrate that the mOS API helps modularize a complex middlebox application by porting Snort3 [16] to using mOS. A typical signature-based NIDS maintains a set of attack signatures (or rules) and examines whether a flow contains the attack patterns. The signatures consist of a large number of rule options that express various attack patterns (e.g., content, pcre), payload type (*e.g.*, http_header) and conditions (e.g., only_stream and to_server). To enhance the modularity of Snort, we leverage the monitoring socket abstraction and express the signatures using event-action pairs.

To transform the signatures into event-action pairs, we express each rule option type as a UDE filter, and synthesize each rule as a chain of UDEs. We use three synthetic rules shown in Figure 10 as an example. For example, the `http_header` rule option in rule (c) corresponds to the filter function FT_{HTTP} that triggers an intermediate event e_{c1}. e_{c1} checks FT_{AC2} for string pattern matching and triggers e_{c2}, which in turn runs PCRE pattern matching (FT_{PCRE}), triggers e_{c3} and finally executes its event handler (f_A). Note, Snort scans the traffic against these rules multiple times: (a) each time a packet arrives, (b) when-

ever enough flow data is reassembled, and (c) whenever a flow finishes. (b) and (c) are required to detect attack patterns that spread over multiple packets in a flow. These naturally correspond to the four mOS built-in events (e_1 to e_4) in Figure 10.

One challenge in the process lies in how one represents the `content` rule option. The `content` rule option specifies a string pattern in a payload, but for efficient pattern matching, it is critical that the payload should be scanned once to find all the string patterns specified by multiple rules. To reflect this need, we use a multi-event filter function that performs Snort's Aho-Corasick algorithm [24]. Given a message, it scans the payload only once and raises distinct events for different string patterns.

We have implemented 17 rule options (out of 45 options[7]) that are most frequently used in Snort rules. Our implementation covers HTTP attack signatures as well as general TCP content attack patterns. The actual implementation required writing a Snort rule parser that converts each rule into a series of UDEs with filter functions so that mSnort3 can run with an arbitrary set of attack signatures. In total, we have modified 2,104 lines out of 79,889 lines of code which replaces the Snort's `stream5` and `http-inspect` modules that provide flow management and HTTP attack detection, respectively.

mSnort3 benefits from mOS in a number of ways. First, we find that each UDE is independent from the internal implementation of flow management, which makes the code easy to read and maintain. Also, the same filter function is reused to define different intermediate UDEs since it can be used to extend a different base event. This makes the rule-option evaluator easier to write, understand, and extend. In contrast, Snort's current rule-option evaluator, which combines the evaluation results of multiple rule options in a rule, is a very complex recursive function that spans over 500 lines of code. Second, since the attack signatures are evaluated in a modular manner, one can easily add new rule options or remove existing ones without understanding the rest of the code. In contrast, such modification is highly challenging with existing `stream5` and `http-inspect` modules since other Snort's modules heavily depend on internal structures and implementation of the two. Third, rules that share the same prefix in the event chain would benefit from sharing the result of event evaluation. Say, two rules are represented as $e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4$, and $e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_5 \rightarrow e_6$. mOS ensures to evaluate up to e_3 only once and shares the result between the two rules. Fourth, mSnort3 can now leverage more fine-grained monitoring features of mOS such as setting a different buffer size per flow or selective buffer management. These features are difficult to implement in the existing code of Snort3.

[7]Remaining options are mostly unrelated to HTTP/TCP protocols.

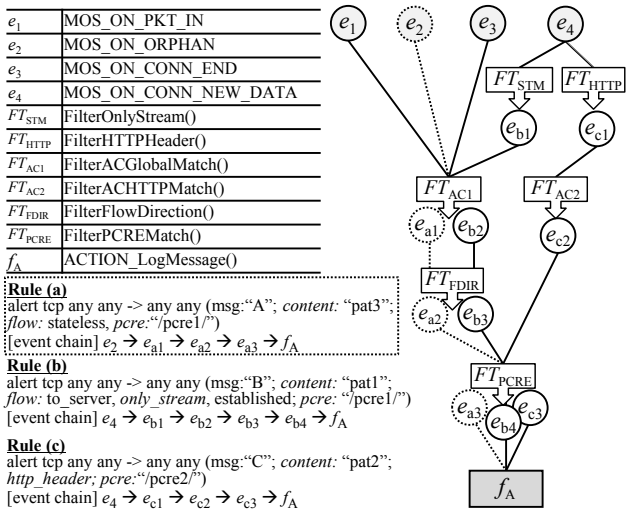


Figure 10: mSnort event-action diagram of sample rules. Keywords in a rule (e.g., 'content' and 'pcre') are translated into a chain of multiple UDEs. String search ('content' option) is performed first. 'only_stream' inspects only flow-reassembled data. The 'msg' string is printed out at f_A if the input data meets all rule options.

mAbacus: The original Abacus code is based on Monbot [63], a home-grown flow monitor for analyzing the content-level redundancy. Abacus reuses the packet I/O and flow management modules of Monbot, while disabling irrelevant features like content hashing. Although the core logic of Abacus is simple, its original implementation requires understanding low-level flow management and modifying 1,880 lines (out of 4,808 lines) of the code. We write mAbacus from a clean slate and in a top-down approach. mAbacus exploits the monitoring socket abstraction to monitor TCP packet retransmission, flow creation/termination and payload arrival events for accounting purpose. Compared to the original version, mAbacus brings two extra benefits. First, it correctly detects retransmission over fragmented segments from out-of-order packets in a receive buffer. Second, one can disable a receive buffer of any side in a single line of code , while original Abacus requires commenting out 100+ lines of its flow processing code manually. The new implementation requires only 561 lines of code. This demonstrates that mOS hides the details of TCP flow management and allows application developers to focus on their own logic.

mHalfback: Halfback [50] is a transport-layer scheme designed for optimizing the flow completion time (FCT). It relies on two techniques: (i) skipping the TCP slow start phase to pace up transmission rate at start, and (ii) performing proactive retransmission for fast packet loss recovery. Inspired by this, we design mHalfback, a middlebox application that performs proactive retransmission over TCP flows. mHalfback has no pacing phase, since a middlebox cannot force a TCP sender to skip the slow start phase. Instead, mHalfback provides fast recovery of any packet

loss, so that it transparen
modification of end-host
back is as follows: (i) v
mHalfback holds a copy
mission. (ii) when a TC
receiver, mHalfback will
certain threshold). mHalf
to hold the packet, and
retransmission. When t
ceeds a retransmission th
it deregisters the packet
any packet beyond the th
ted. Likewise, mHalfbac
its connection is closed.
tected with the state cha
of mOS monitoring soc
requires only 128 lines o

mnDPI library: nDPI [9] is an open-source DPI library that detects more than 150 application protocols. It scans each packet payload against a set of known string patterns or detects the protocol by TCP/IP headers. Unfortunately, it neither performs flow reassembly nor properly handles out-of-order packets. We have ported nDPI (`libndpi`) to use the mOS API by replacing their packet I/O and applying UDEs derived from 4 built-in events as in mSnort3. Our porting enables all applications that use `libndpi` to detect the patterns over flow-reassembled data. This requires adding 765 lines of code to the existing 25,483 lines of code.

mPRADS: PRADS [12] is a passive fingerprinting tool that detects the types of OSes and server/client applications based on their network traffic. It relies on PCRE [13] pattern matching on TCP packets for this purpose. Like nDPI, PRADS does not perform flow reassembly. Furthermore, despite its comprehensive pattern set, the implementation is somewhat ad-hoc since it inspects only the first 10 (which is an arbitrarily set threshold) packets of a flow. mPRADS employs 24 UDEs on `MOS_ON_CONN_NEW_DATA` to detect different L7 protocols in separate event handlers. This detects the patterns regardless of where the pattern appears in a TCP connection. We modify only 615 lines out of 10,848 lines to update mPRADS.

5.2 mOS Performance

We now evaluate the performance of mOS, including the flow management and event system.

Experiment setup: We evaluate mOS applications on a machine with dual Intel E5-2690 v0 CPUs (2.90GHz, 16 cores in total), 64 GB of RAM, and 3 dual-port Intel 82599 10G NICs. For flow generation, we use six pairs of clients and servers (12 machines) where each pair communicates via a 10G link through an mOS application. Each client/server machine is equipped with one Intel Xeon E3-1220 v3 CPU (4 core, 3.10 GHz), 16 GB of RAM, and a Intel 10G NIC. All machines run Linux kernel version 3.13 and use Intel's DPDK v16.04.

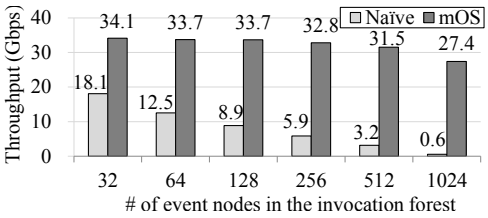


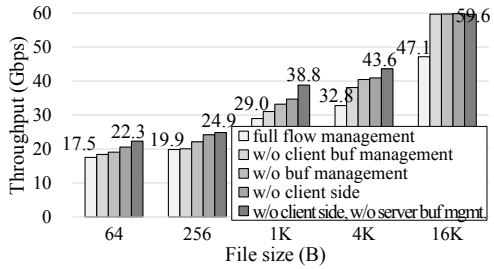
Figure 12: Performance at dynamic event registration

Synthetic workload: The client spawns many concurrent flows that download HTTP objects of the same size from a server. When an object download completes, the client fetches another object so that the number of concurrent connections stays the same. Both the client and server are implemented with mTCP [43] for high performance, and all six pairs can generate up to 192K concurrent flows.

Microbenchmarks: Figure 11(a) shows the performance of mOS applications when the clients fetch HTTP objects (64 bytes or 8 KB) with 192K concurrent connections. `packetcount` counts the number of packets per each flow (by using a `MOS_ON_PKT_IN` event handler) while `stringsearch` performs keyword string search over flow-reassembled data (by using a `MOS_ON_CONN_NEW_DATA` event handler). We observe that the performance almost linearly scales over the number of CPU cores, and both applications achieve high throughputs despite a large number of concurrent flows when multiple CPU cores are employed. At 16 cores, `packetcount` produces 19.1 Gbps and 53.6 Gbps for 64 bytes and 8 KB objects, respectively while `stringsearch` achieves 18.3 Gbps and 44.7 Gbps for 64 bytes and 8 KB, respectively. Figure 11(b) shows the flow completion time for the two mOS applications. mOS applications add 41 $\sim$ 62 us of delay to that of a direct connection (without any middlebox) for 64-byte objects and 81 $\sim$ 170 us of latency for 8 KB objects. We believe the latency stretch is reasonable even when a middlebox operates in the middle.

Figure 11(c) compares the performances of application `packetcount` under various levels of resource consumption. Depending on the file size, selective resource configuration improves the performance by up to 25% to 34% compared with the full flow management of both sides. Not surprisingly, disabling the entire state update of one side provides the biggest performance boost, but skipping flow buffering also brings non-trivial performance improvement. This confirms that tailoring resource consumption to the needs of a specific middlebox application produces significant performance benefit.

Efficient i-forest management: Figure 12 compares the performance of `stringsearch` as it dynamically registers for an event. Clients download 4KB objects with 192K concurrent flows. When the application finds the



(c) Performance under selective resource consumption

Figure 11: Microbenchmark experiments with a standalone middlebox.

target string, it registers for an extra event with the socket to inspect the data further. We have the server inject the target string for 50% of object downloads, and vary the number of initial registered events per socket from 32 to 1024. A naïve implementation would take a snapshot of the invocation forest and copy the entire forest to dynamically register for an event. In contrast, our algorithm looks up the same invocation forest and spawns a new one only if it does not exist. Our algorithm outperforms the naïve implementation by 15.4 to 26.8 Gbps depending on the number of event nodes, which confirms that fast identification of the same forest greatly enhances the performance at dynamic event registration.

Memory footprint analysis: Each TCP context takes up 456 bytes of metadata and starts with a 8 KB flow buffer. So, 192K concurrent flows would consume about 3.2 GB with bidirectional flow reassembly. For i-forest management, an event node takes up 128 bytes. Thus, monitoring 4,000 events per connection (e.g., Snort) would consume n * 512 KB, if the application ends up generating n distinct i-forests for updating events for k times during its operation (typically, $k >> n$). In contrast, a naïve implementation would consume k * 512 KB, which would use 98 GB if all 192K flows update their i-forest dynamically ($k = 192K$).

Handling abnormal traffic: We evaluate the behavior of mOS when it sees a large number of abnormal flows. We use the same test environment, but have the server transmit the packets out of order (across the range of sender's transmission window) at wire rate. We confirm that (a) mOS correctly buffers the bytestream in the right order, and (b) its application shows little performance degradation from when clients and servers directly communicate.

5.3 mOS Application Evaluation

We verify the correctness of mOS-based middleboxes and evaluate their performance with real traffic traces.

Correct flow reassembly: We test the correctness of mnDPI, mPRADS, and mAbacus under lightweight load with 24K concurrent flows, downloading 64KB HTTP objects. We randomly inject 10 different patterns where each pattern crosses over 2 to 25 packets in different flows and see if mnDPIReader (a simple DPI application using libndpi) and mPRADS detect them. We repeat the test for 100 times, and confirm that mnDPI and mPRADS successfully detect the flows while their original versions miss them. We also inject 1K fake retransmission flows and find that mAbacus successfully detects all such attacks.

Application	original + pcap	original + DPDK	mOS port
Snort-AC	0.51 Gbps	8.43 Gbps	9.85 Gbps
Snort-DFC	0.78 Gbps	10.43 Gbps	12.51 Gbps
nDPIReader	0.66 Gbps	29.42 Gbps	28.34 Gbps
PRADS	0.42 Gbps	2.05 Gbps	2.02 Gbps
Abacus	-	-	28.48 Gbps

Table 3: Performance of original and mOS-ported applications under a real traffic trace. Averaged over five runs.

Performance under real traffic: We measure the performance of original applications and their mOS ports with a real network packet trace. The trace is obtained from a large cellular ISP in South Korea and records 89 million TCP packets whose total size is 67.7 GB [63]. It contains 2.26 million TCP connections and the average packet size is 760 bytes. We replay the traffic at the speed of 30 Gbps to gauge the maximum performance of mOS-ported applications. We use the same machines as in Section 5.2.

Table 3 compares the performances of Snort, nDPIReader, PRADS, and Abacus. Snort-DFC uses a more efficient multi-string matching algorithm, DFC [33], instead of the Aho-Corasick algorithm (Snort-AC). Original applications (except Abacus) use the pcap library by default, which acts as the main performance barrier. Porting them to use the DPDK library greatly improves the performance by a factor of 4.9 to 44.6 due to scalable packet I/O. mOS-based applications deliver comparable performances to those of DPDK-ports while mOS ports provide code modularity and correct operation in pattern matching. This confirms that mOS does not incur undesirable performance overhead over DPDK-ported applications.

mSnort is actually slightly faster than Snort+DPDK. This is mainly due to the improved efficiency of mOS's flow management over Snort's `stream5`. Our profiling finds that the `stream5` module incurs more memory accesses per packet on average. Compared to others, PRADS shows much lower performance because it naïvely performs expensive PCRE pattern matching on the traffic.

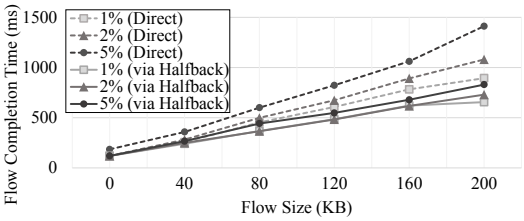


Figure 14: Average flow completion time by packet loss ratio

The flow management overhead in mnDPIReader and mPRADS is small, representing only about 1 to 4% of performance change.

Performance under packet loss: We evaluate mHalfback by injecting Web traffic into a lossy network link. We test with the same topology used in the original paper [50] as shown in Figure 13. Figure 14 compares average FCT of a direct connection (Figure 13(A)) and of a connection via Halfback proxy (Figure 13(B)) under various packet loss rates. We find that mHalfback significantly reduces the FCT with the help of fast loss recovery. When testing with flows that download 100 KB under 5% packet loss, mHalfback brings 20% to 41% FCT reduction. While this is lower than 58% FCT reduction reported by the original paper, it is promising as it does not require any modification on the server.

6 Related Work

We discuss previous works that are related to mOS.

Flow management support: `libnids` [8] is a flow-level middlebox library that can be used for building middlebox applications but it does not provide comprehensive flow reassembly features [17, 22]. Bro [55] provides an event-driven scripting language for network monitoring. mOS differs from Bro in its programming model. mOS is designed to write broader range of network applications and provides an API that allows more fine-grained control over live connections. A mOS middlebox developer can *dynamically* register new events per flow at any stage of a connection life cycle, a flow's TCP receive buffer management can be disabled at run-time, and monitoring of any side (client or server) of the flow can be disabled dynamically. Bro does not offer such features.

Modular middlebox development: Click [46] provides a modular packet processing platform, which allows development of complex packet forwarding applications by chaining *elements*. Click has been a popular platform in research community to implement L3 network function prototypes [26, 29, 45, 49, 52]. mOS, on the other hand, provides comprehensive, flexible flow-level abstractions that allow mapping a custom flow-level *event* to a corresponding *action*, and is suitable for building L4-L7 monitoring applications. CliMB [48] provides a modular TCP layer composed of Click elements, but its TCP-based elements are designed only for end-host stacks; whereas mOS facilitates programming middleboxes.

xOMB [25] is a middlebox architecture that uses programmable pipelines that simplify the development of inline middleboxes. Although xOMB shares the goal of simplifying development of flow-level middleboxes, its system is focused on an L7 proxy, which uses BSD sockets to initialize and terminate connections. mOS focuses on exposing flow-level states and events for general-purpose monitoring applications without the ability to create new connections. CoMb [59] aims to provide efficient resource utilization by consolidating common processing actions across multiple middleboxes; while mOS cuts down engineering effort by combining common flow-processing tasks on a single machine dataplane.

Scalable network programming libraries: Several high-speed packet I/O frameworks have been proposed [4, 14, 40, 57]. However, extending these frameworks to support development of stateful middlebox applications requires significant software engineering effort. mOS uses mTCP [43], a scalable user-level multicore TCP stack for stateful middleboxes. The performance scalability of mOS comes from mTCP's per-thread socket abstraction, shared-nothing parallel architecture, and scalable packet I/O. IX [28] and Arrakis [56] present new networking stack designs by separating the kernel control planes from data planes. However, both models only provide endpoint networking stacks. There are a few on-going efforts that provide fast-path networking stack solutions [1, 11, 15, 21] for L2/L3 forwarding data planes. We believe mOS is the first fast-path networking stack which provides comprehensive flow-level monitoring capabilities for L4-L7 stateful middleboxes. Modnet [54] provides modular TCP stack customization for demanding applications, but only for end host stack.

7 Conclusion

Modular programming of stateful middleboxes has long been challenging due to complex low-level protocol management. This works addresses the challenge with a general-purpose, reusable networking stack for stateful middleboxes. mOS provides clear separation of interface and implementation in building complex stateful middleboxes. Its flow management module provides accurate tracking of the end-host states, enabling the developer to interact with the system with a well-defined set of APIs. Its flow event system flexibly expresses per-flow conditions for custom actions. The mOS source code is available at `https://github.com/ndsl-kaist/mos-networking-stack`.

8 Acknowledgments

We would like to thank our shepherd Minlan Yu and anonymous reviewers of NSDI'17 for their insightful comments on the paper. We also thank Shinae Woo and Keon Jang for their contributions on the early design of the mOS networking stack. This work was supported in part by the ICT R&D Program of MSIP/IITP, Korea, under Grants B0126-16-1078, B0101-16-1368 [Development of an NFV-inspired networked switch and an operating system for multi-middlebox services],[2016-0-00563, Research on Adaptive Machine Learning Technology Development for Intelligent Autonomous Digital Companion], and by the National Research Council of Science & Technology (NST) grant by the Korea government (MSIP) No. CRC-15-05-ETRI.

References

[1] 6WINDGate Software: Network Optimization Software. http://www.6wind.com/products/6windgate/.

[2] Accelerating Snort with PF_RING DNA – ntop. http://www.ntop.org/pf_ring/accelerating-snort-with-pf_ring-dna/.

[3] Argus - NSMWiki. http://nsmwiki.org/index.php?title=Argus.

[4] Intel DPDK: Data Plane Development Kit. http://dpdk.org/.

[5] IPTABLES manpage. http://ipset.netfilter.org/iptables.man.html.

[6] Kismet Wireless Network Detector. https://www.kismetwireless.net/.

[7] L7-filter: Application Layer Packet Classifier for Linux. http://l7-filter.sourceforge.net/.

[8] Libnids. http://libnids.sourceforge.net/.

[9] nDPI | ntop. http://www.ntop.org/products/ndpi/.

[10] netfilter/iptables project homepage. http://www.netfilter.org/.

[11] OpenFastPath: Technical Overview. http://www.openfastpath.org/index.php/service/technicaloverview/.

[12] Passive Real-time Asset Detection System. http://gamelinux.github.io/prads/.

[13] PCRE - Perl Compatible Regular Expressions. http://www.pcre.org/.

[14] PF_RING ZC (Zero Copy). http://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/.

[15] Project Proposals/TLDK. https://wiki.fd.io/view/Project_Proposals/TLDK.

[16] Snort 3.0. https://www.snort.org/snort3.

[17] Snort: Re: Is there a snort/libnids alternative. http://seclists.org/snort/2012/q4/396.

[18] Snort: Reputation Preprocessor. http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node17.html#SECTION00322000000000000000.

[19] Suricata Open Source IDS/IPS/NSM engine. http://suricata-ids.org/.

[20] Tcpdump & Libpcap. http://www.tcpdump.org/.

[21] The Fast Data Project (FD.io). https://fd.io/.

[22] wireshark - using "follow tcp stream"; code in my c project - Stack Overflow. https://ask.wireshark.org/questions/39531/using-follow-tcp-stream-code-in-my-c-project.

[23] YouTube. https://www.youtube.com/.

[24] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, June 1975.

[25] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: extensible open middleboxes with commodity servers. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2012.

[26] B. Anwer, T. Benson, N. Feamster, and D. Levin. Programming Slick Network Functions. In *Proceedings of the ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, 2015.

[27] T. Barbette, C. Soldani, and L. Mathy. Fast Userspace Packet Processing. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2015.

[28] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[29] A. Bremler-Barr, Y. Harchol, and D. Hay. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2016.

[30] Bug #1238: Possible evasion in stream-reassemble.c. Suricata-3.11 ChangeLog. https://github.com/inliniac/suricata/blob/master/ChangeLog.

[31] Bug #1557: Stream: retransmission not detected. Suricata-3.11 ChangeLog. https://github.com/inliniac/suricata/blob/master/ChangeLog.

[32] Bug #1684: eve: stream payload has wrong direction in IPS mode. Suricata-3.11 ChangeLog. https://github.com/inliniac/suricata/blob/master/ChangeLog.

[33] B. Choi, J. Chae, M. Jamshed, K. Park, and D. Han. DFC: Accelerating Pattern Matching for Network Applications. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[34] Common Vulnerabilities and Exposures. CVE - CVE-2003-0209: DSA-297-1 Snort – integer overflow. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0209.

[35] Common Vulnerabilities and Exposures. CVE - CVE-2004-2652: Snort 2.3.0 – DecodeTCP Vulnerabilities. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0209.

[36] Common Vulnerabilities and Exposures. CVE - CVE-2009-3641: Snort 2.8.5 IPv6 Remote DoS. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3641.

[37] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2009.

[38] ETSI. Network Functions Virtualization – Introductory White Paper. https://portal.etsi.org/NFV/NFV_White_Paper.pdf, 2012.

[39] Y. Go, E. Jeong, J. Won, Y. Kim, D. F. Kune, and K. Park. Gaining Control of Cellular Traffic Accounting by Spurious TCP Retransmission. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[40] S. Han, K. Jang, K. Park, and S. B. Moon. PacketShader: a GPU-Accelerated Software Router. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2010.

[41] M. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park. Kargus: a Highly-scalable Software-based Intrusion Detection System. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[42] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: cheap SSL acceleration with commodity processors. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[43] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[44] H. Jiang, G. Zhang, G. Xie, K. Salamatian, and L. Mathy. Scalable High-Performance Parallel Design for Network Intrusion Detection Systems on Many-Core Processors. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2013.

[45] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller. Stateless Network Functions. In *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, 2015.

[46] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[47] H. Krawczyk. LFSR-based Hashing and Authentication. In *Proceedings of the Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 1994.

[48] R. Laufer, M. Gallo, D. Perino, and A. Nandugudi. CliMB: Enabling Network Function Composition with Click Middleboxes. In *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, 2016.

[49] B. Li, K. Tan, L. Luo, R. Luo, Y. Peng, N. Xu, Y. Xiong, P. Chen, Y. Xiong, and P. Cheng. ClickNP: Highly Flexible and High-performance Network Processing with Reconfigurable Hardware. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2016.

[50] Q. Li, M. Dong, and P. B. Godfrey. Halfback: Running Short Flows Quickly and Safely. In *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2015.

[51] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider. Enriching Network Security Analysis with Time Travel. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2008.

[52] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[53] Novak, Judy and Sturges, Steve. Target-Based TCP Stream Reassembly. *Sourcefire Inc.*, pages 1–23, 2007.

[54] S. Pathak and V. S. Pai. ModNet: A Modular Approach to Network Stack Extension. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[55] V. Paxson. Bro: A System for Detecting Network Intruders in Real-time. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 1998.

[56] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2014.

[57] L. Rizzo. netmap: a novel framework for fast packet I/O. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2012.

[58] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the USENIX Systems Administration Conference (LISA)*, 1999.

[59] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[60] The Bro Project. Writing Bro Plugins. https://www.bro.org/sphinx/devel/plugins.html.

[61] The Snort Project. Snort Users Manual 2.9.5. pages 59–73, 2013.

[62] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. MIDeA: A Multi-Parallel Intrusion Detection Architecture. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[63] S. Woo, E. Jeong, S. Park, J. Lee, S. Ihm, and K. Park. Comparison of Caching Strategies in Modern Cellular Backhaul Networks. In *Proceeding of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2013.

Appendix A

```
/* monitoring socket creation/closure, scope setup (see Figure 2) */
int mtcp_socket(mctx_t mctx, int domain, int type, int protocol);
  — Create a socket.  The socket can be either regular TCP socket (type = MOS_SOCK_STREAM),
  — a TCP connection monitoring socket (type = MOS_SOCK_MONITOR_STREAM) or a raw packet monitoring socket (type = MOS_SOCK_MONITOR_RAW).
int mtcp_bind_monitor_filter(mctx_t mctx, int sock, monitor_filter_t ft);
  — Bind a monitoring socket(sock) to a traffic filter (ft).  ft limits the traffic monitoring scope in a BPF syntax.
int mtcp_getpeername(mctx_t mctx, int sock, struct sockaddr *addr, socklen_t *addrlen);
  — Retrieve the peer address information of a socket (sock).
  — Depending on the size of addrlen, one can get either server-side or both server and client-side address information.
int mtcp_close(mctx_t mctx, int sock);
  — Close a socket.  Closing a monitoring socket does not terminate the connection but unregisters all flow events for the socket.
/* event manpulation (see Figure 2 and Section 3.2) */
event_t mtcp_define_event(event_t ev, filter_t filt, struct filter_arg *arg);
  — Define a new event with a base event (ev) and a filter function (filt) with a filter argument (arg).
event_t mtcp_alloc_event(event_t parent_event); /
int mtcp_raise_event(mctx_t mctx, event_t child_event);
  — Define a follow-up event for a filter that can trigger multiple (child) events.
  — Raise a child event for a multi-event filter.
int mtcp_register_callback(mctx_t mctx, int sock, event_t ev, int hook, callback_t cb);
  — Register (or unregister) an event handler (or a callback function) (cb) for an event (ev) in the context of a monitoring socket (sock).
  — hook specifies when the event should be fired.  It can be fired after updating packet sender's TCP context (MOS_HK_SND)
  — or after updating packet receiver's TCP context (MOS_HK_RCV) or MOS_NULL, which does not care.
/* current packet information and modification (see Figure 4) */
int mtcp_getlastpkt(mctx_t mctx, int sock, int side, struct pkt_info *pinfo); /
int mtcp_setlastpkt(mctx_t mctx, int sock, int side, off_t offset, byte *data, uint16_t datalen, int option);
  — mtcp_getlastpkt() retrieves the information of the last packet of a flow (sock and side).
  — mtcp_setlastpkt() updates the last packet with data at offset bytes from an anchor for datalen bytes.
  — option is the anchor for offset.  It can be one of MOS_ETH_HDR, MOS_IP_HDR, MOS_TCP_HDR or MOS_TCP_PAYLOAD.
int mtcp_sendpkt(mctx_t mctx, int sock, const struct pkt_info *pkt);
  — Send a self-constructed TCP packet (pkt) for a given flow (sock).
/* flow-reassembled buffer reading (see Figure 4 and Section 4.3) */
ssize_t mtcp_peek(mctx_t mctx, int sock, int side, char *buf, size_t len); /
ssize_t mtcp_ppeek(mctx_t mctx, int sock, int side, char *buf, size_t count, off_t seq_off);
  — Read the data in a TCP receive buffer of sock.  side specifies either client or server side.
  — mtcp_ppeek() is identical to mtcp_peek() except that it reads the data from a specific offset(seq_off) from the initial sequence number.
/* TCP flow monitoring and manipulation (see Figures 4 & 7) */
int mtcp_getsockopt(mctx_t mctx, int sock, int level, int optname, void *optval, socklen_t *optlen); /
int mtcp_setsockopt(mctx_t mctx, int sock, int level, int optname, void *optval, socklen_t optlen);
  — Retrieve (or set) socket-level attributes.
/* per-flow user-level metadata management */
void *mtcp_get_uctx(mctx_t mctx, int sock); /
void mtcp_set_uctx(mctx_t mctx, int sock, void *uctx);
  — Retrieve (or store) a user-specified pointer (uctx) associated with a socket (sock).
/* initialization routines */
mctx_t mtcp_create_context(int cpu); /
int mtcp_destroy_context(mctx_t mctx);
  — Create (or destroy) a mOS context and associate it with a cpu id.
int mtcp_init(const char *mos_conf_fname);
  — Initialize mOS with the attributes in a configuration file (mos_conf_fname).  Called one time per process.
```

Table 4: The current mOS networking API. More detail is found in mOS manual pages: http://mos.kaist.edu/index_man.html.

Appendix B

```
// count # of packets in each TCP flow
static void // callback for MOS_ON_PKT_IN
OnFlowPkt(mctx_t m, int sock, int side, event_t event,
          struct filter_arg *arg)
{
   if (side == MOS_SIDE_CLI)
      g_pktcnt[sock]++;
}

// count # of packet retransmissions in each TCP flow
static void // callback for MOS_ON_REXMIT
OnRexmitPkt(mctx_t m, int sock, int side, event_t event,
            struct filter_arg *arg)
{
   g_rexmit_cnt[sock]++;
}

// count # of client-initated TCP connection teardown
static void // callback for MOS_ON_TCP_STATE_CHANGE
OnTCPStateChange(mctx_t m, int sock, int side, event_t event,
                 struct filter_arg *arg)
{
   if (side == MOS_SIDE_CLI) {
      int state; socklent_t len = sizeof(state);
      mtcp_getsockopt(m, sock, SOL_MONSOCKET,
                      MOS_TCP_STATE_CLI, &state, &len);
      if (state == TCP_FIN_WAIT_1)
         g_cli_term++;
   }
}

// print the statistics and reset counters
// count total # of completed flows
static void // callback for MOS_ON_TCP_CONN_END
OnFlowEnd(mctx_t m, int sock, int side, event_t event,
          struct filter_arg *arg)
{
   if (sock != MOS_SIDE_CLI) return;

   TRACE_LOG("TCP flow (sock=%d) had %d packets, rexmit: %d\n",
             sock, g_pktcnt[sock], g_rexmit_cnt[sock]);
   g_pktcnt[sock] = 0; g_rexmit_cnt[sock] = 0;
   g_total_flows++;
}
```

Figure 15: Code examples with mOS built-in event handlers. With only 2∼5 lines of code, one can gather various flow-level statistics in a middlebox.

One Key to Sign Them All Considered Vulnerable: Evaluation of DNSSEC in the Internet

Haya Shulman and Michael Waidner
Fraunhofer Institute for Secure Information Technology SIT

Abstract

We perform the first Internet study of the cryptographic security of DNSSEC-signed domains. To that end, we collected $2.1M$ DNSSEC keys for popular signed domains out of these $1.9M$ are RSA keys. We analyse the RSA keys and show that a large fraction of signed domains are using vulnerable keys: 35% are signed with RSA keys that share their moduli with some other domain and 66% use keys that are too short (1024 bit or less) or keys which modulus has a GCD > 1 with the modulus of some other domain. As we show, to a large extent the vulnerabilities are due to poor key generation practices, but also due to potential faulty hardware or software bugs.

The DNSSEC keys collection and analysis is performed on a daily basis with the *DNSSEC Keys Validation Engine* which we developed. The statistics as well as the DNSSEC Keys Validation Engine are made available online, as a service for Internet users.

1 Introduction

Domain Name System (DNS), [RFC1034, RFC1035], is one of the Internet's key protocols, designed to locate resources in the Internet. The correctness and availability of DNS are critical to the security and functionality of the Internet. Initially designed to translate domain names to IP addresses, the DNS infrastructure has evolved into a complex ecosystem, and the complexity of the DNS infrastructure is continuously growing with the expanding range of purposes and client base. DNS is increasingly utilised to facilitate a wide range of applications and constitutes an important building block in the design of scalable network infrastructures.

There is a long history of attacks against DNS, most notably, DNS cache poisoning, [41, 29, 23, 39, 38, 26, 25, 24]. DNS cache poisoning attacks are known to be practiced by governments, e.g., for censorship [1] or for surveillance [28], as well as by cyber criminals. In the course of a DNS cache poisoning attack, the attacker provides spoofed records in DNS responses, in order to redirect the victims to incorrect hosts for credentials theft, malware distribution, censorship and more. To prevent such attacks a number of non-cryptographic defences were standardised [RFC5452].

A number of works developed techniques to bypass the (non cryptographic) DNS defences against cache poisoning. For instance, [30, 25], showed how to predict the source port and transaction identifier (TXID) values used by DNS resolver in DNS requests. The attacker can also apply fragmentation [24] to circumvent the challenge values that would remain in the first fragment. These methods allow off-path attacker to inject valid DNS responses into the communication between a victim DNS resolver and a nameserver. Of course, a man-in-the-middle (MitM) attacker does not need to guess the challenge values: a MitM attacker sees the DNS requests and hence can simply copy the challenge-response authentication parameters from the request to the response. Domains can also be hijacked by exploiting misconfigurations in DNS records, for instance, via manipulation of dangling DNS records (*Dare*), [34].

Recently, [31] performed a study of the DNS caches, evaluating different approaches for injecting DNS records into caches, and overwriting the "already cached" values. In particular, [31] evaluated which records could overwrite the already cached values in victim DNS caches, that would further provide them to the clients. The model of caches was then applied to DNS resolution platforms in the wild, and [31] found that 97% of open resolvers, almost 70% of DNS caches on ISP networks and a bit more than 70% of DNS caches on enterprise networks were vulnerable to at least one type of DNS records injection attacks. The question whether cache poisoing actually occurs in the Internet motivated measurement works which evaluated the fraction of spoofed DNS records in the wild [32, 36, 44].

To mitigate DNS cache poisoning attacks, the IETF designed and standardised Domain Name System Security Extensions (DNSSEC) [RFC4033-RFC4035]. Due to the changes to the DNS infrastructure and to the DNS protocol that DNSSEC requires, its adoption is progressing slowly. Although proposed and standardised already in 1997, DNSSEC is still not widely deployed. Measurements show that currently about 25% of the DNS resolvers validate DNSSEC (e.g., see `stats.labs.apnic.net/dnssec`), which is an increase since few years ago, where about 3% of the DNS resolvers were measured to validate DNSSEC records, [33, 14]. Although only a bit more than 1% of domains are signed with DNSSEC, [47, 27], the situation is improving. More domains are expected to get signed in the near future. This is due to the fact that domains signing is turning into an easy task, as the procedures for automated domains signing by the registrars and hosting providers are becoming widely supported. In particular, ICANN requires that all its accredited registrars and DNS hosting providers support domains signing, encouraging adoption of automated signing procedures, where the DNSSEC keys are generated for customers' domains by the provider and used to sign the DNS records. This process assumes that the registrars and DNS hosting providers are supporting secure and best practices for keys generation.

In this work we perform a study of the vulnerabilities of DNSSEC keys generation in *signed domains*. We detect vulnerabilities and trace the problems to the DNSSEC keys generation practices used by the registrars and DNS hosting providers. The main cause for the problems is twofold: *saving on randomness or faults* during keys generation and *lack of suitable key validation procedures*. The vulnerabilities that we found are not detectable by the online DNSSEC checkers, such as `dnssec-debugger` of Verisign[1] or `DNSViz` of Sandia National Laboratories[2], or DNSSEC command line validation procedures for zonefiles signing.

Our study shows that well established and popular DNS hosting providers and registrars, including those commended by ICANN[3], generate 'weak' keys, for the domains that they host.

Our work is related to a recent study of [22] which showed vulnerabilities in TLS and SSH keys, generated by headless or embedded systems, and server management cards. Subsequently, [22] concluded that the problems are due to faulty implementations that generate keys automatically on first boot without having collected sufficient entropy, and that the problems would not occur on traditional PCs. In contrast, we find vulnerabilities in key generation procedures used by well established hosting providers and registrars (most of which are ICANN accredited[4]), which have the necessary infrastructure to produce the randomness needed for secure generation of DNSSEC keys.

Ultimately our work provides yet another proof that adoption of cryptography in the Internet is a challenging task in practice.

Contributions

In this work we show that even DNSSEC, which is a relatively straightforward application of digital signatures over sets of DNS records, is incurring practical issues in the implementation. This indicates that adoption of cryptography in the Internet is a challenging and error prone task, which requires careful designs and most importantly automation.

We identified vulnerabilities in DNSSEC key generation mechanisms, which exposes signed domains to attacks. To evaluate the scope of the vulnerabilities and the status of DNSSEC adoption we design and implement a framework, we call *DNSSEC keys validation engine*, which allows to collect DNSSEC keys from multiple sources, analyse their security and process them into reports. Our analysis focuses on RSA keys, which as we show in Section 5, constitute a vast majority of the deployed DNSSEC cryptographic algorithms. Our reports quantify the following types of vulnerabilities in signed domains: *keys with shared modulus*, *shared keys* and *weak keys*. The former two categories are comprised of all the signed domains which keys share RSA moduli, either due to use of the same key pair (N,e,d) for all domains or due use of a shared moduli N and different (e,d). The latter category contains signed domains whose keys moduli have a prime factor with 'vulnerable' GCD, i.e., greater than 1 or even, or domains with short (weak) RSA keys. We use our engine to perform Internet-wide collection of $2.1M$ DNSSEC keys[5] used by $900K$ signed domains. We focus on $1.9M$ RSA keys and show that 35% share RSA moduli or are used to sign multiple domains, and 66% fall in the category of 'weak keys', namely, are too short (1024 bit or less) with 0.4% keys being shorter than 768 bit, or have an even GCD. In Section 6.1 we show how to recover the secret signing keys for signed domains which share RSA moduli. In Section 6.2 we apply the fast pairwise GCD algorithm (implemented by [22] and available at `http://factorable.net/`) for all the RSA public keys col-

[1] `dnssec-debugger.verisignlabs.com`

[2] `dnsviz.net`

[3] `https://www.icann.org/resources/pages/deployment-2012-02-25-en`

[4] `http://www.internetsociety.org/deploy360/resources/dnssec-registrars/`

[5] In this work, we report on analysis of keys we collected over a period of March 2016 and September 2016.

lected in order to check whether any of the moduli share a common factor.

We analyse our findings and trace the problems to key generation practices by popular registrars and hosting providers. We discuss good and bad practices for keys generation and derive recommendations based on our study.

We created an online webpage with a list of vulnerable signed domains and DNSSEC keys at `www.dnssec.sit.fraunhofer.de/` (the GitHub project with the code and the data is linked to from the website). Our tool enables the clients to identify secure registrars for signing their domains and alert in case of potential vulnerabilities. We also created an online keys validation service which allows to establish security of the keys during the generation phase, before they are published in public registries and used to sign DNS records.

Organisation

In Section 2 we compare our research to related work. In Section 3 we provide background on DNS and DNSSEC, recap RSA definition and describe two attacks, relevant to our work. In Section 4 we present our DNSSEC-keys validation engine, its components and the data collection that we performed. In Section 5 we perform a measurement of signed domains, then in Section 6 we validate these domains for vulnerable keys, and characterise the factors causing vulnerabilities. We provide countermeasures and describe implementation thereof in Section 7. We conclude this work in Section 8.

2 Related Work

Our work is related to studies on measuring adoption of DNSSEC in the Internet and evaluation of vulnerable cryptography, most notably SSL/TLS. We first compare our evaluation of vulnerable cryptography to the studies conducted in prior work, and then review earlier measurements of adoption of DNSSEC.

Security of cryptographic systems depends on the randomness used in keys generation process. There is a long history of attacks exploiting insufficient randomness in current random number generators, e.g., [5, 7, 9, 12, 20, 21, 16]. There were also attacks applied against systems relying on sources of randomness, e.g., insufficient randomness in pseudorandom number generator was exploited in [30] to predict the TXID values selected by Bind implementations. In 2008, [46] observed that due to an implementation bug the pseudorandom number generator of Debian OpenSSL was predictable.

Adoption of cryptography in the Internet is similarly a challenging task. For instance, the most widely adopted cryptographic mechanism SSL/TLS experienced many attacks due to different implementation faults, e.g., [13, 2]. See [35] for a comprehensive review of vulnerabilities in SSL/TLS.

Most related to our work, is Heninger *et al.* [22] that performed an Internet-wide scan of TLS certificates and SSH keys to analyse security of random number generators (RNGs), and found vulnerabilities in about 1% of TLS certificates and SSH hosts. They showed that the vast majority of the vulnerable hosts are headless and embedded devices, which lack sufficient randomness. In contrast, in our work, we study the registrars and DNS hosting providers, which have the required infrastructure to generate the necessary randomness, yet the fraction of the vulnerable keys in DNSSEC-signed zones that we find is significantly larger than that measured by [22] in keys in TLS and SSH. The results of our work extend the conclusion of [22], in particular, we show that the problem with randomness is not inherent to restricted embedded devices, but is more significant – we show that even large and popular DNS hosting providers and registrars introduce vulnerabilities into multiple signed domains. Our work provides evidence that the problem with the randomness is not only due to resources limited devices but is wider and applies to platforms which possess the necessary means to produce the randomness required for security.

[4] performed factorisation of 1024-bit RSA keys. Some of the keys were vulnerable due to sharing of primes which allowed efficient factorisation via batch GCD computation, while others were factored by taking advantage of randomness generation process.

Previous work on DNSSEC adoption in domains measured the fraction of the *signed* domains, [47, 27], or misconfigurations in signed domains [10, 11], which result in degraded availability. The client side of the DNS infrastructure was also studied, mainly measuring the fraction of *validating resolvers*, [33, 19]. Zone enumeration against NSEC3 was first performed by Bernstein [3] and then Wander *et al.* [45]. Recently Goldberg *et al.* [18] showed that suggestions to improve NSEC3 were also vulnerable to zone enumeration, and proposed [17].

The challenge of performing large scale active measurements of DNS were discussed in [43], which designed and developed an infrastructure for collecting and analysing DNS packets from multiple domains. Due to the large traffic volume that the measurement infrastructure produced (e.g., 123M domains in `com`) and the requirement for repeated data collection on a daily basis, [43] had different latency and storage considerations, than we in our work. In particular, we only focus on $900K$ signed domains. In contrast to [43] we process the results and display them in reports.

3 DNS Security with DNSSEC

In this section we review DNS, and DNSSEC, and then describe the RSA cryptosystem used in DNSSEC and attacks against it (that are relevant to our work).

3.1 Domain Name System (DNS)

Domain Name System (DNS), [RFC1034, RFC1035], is a distributed database containing mappings for resources (also called *resource records (RRs)*), from *domain names* to different values. The most popular and widely used mappings, [15], are for IP addresses, represented by `A` type RRs, that map a domain name to its IPv4 address, and name servers, represented by `NS` type RRs, that map a name server to domain name. The resource records in DNS correspond to the different services run by the organisations and networks, e.g., hosts, servers, network blocks.

The zones are structured hierarchically, with the root zone at the first level, Top Level Domains (TLDs) at the second level, and millions of Second Level Domains (SLDs) at the third level. The IP addresses of the 13 root servers are provided via the *hints* file, or compiled into DNS resolvers software and when a resolver's cache is empty, every resolution process starts at the root. According to the query in the DNS request, the root name server redirects the resolver, via a `referral` response type, to a corresponding TLD, under which the requested resource is located. There are a number of TLDs types, most notably: *country code TLD* (ccTLD), which domains are (typically) assigned to countries, e.g., `us, il, de`, and *generic TLD* (gTLD), whose domains are used by organisations, e.g., `com, org`, and also by US government and military, e.g., `gov, mil`. Domains in SLDs can also be used to further delegate subdomains to other entities, or can be directly managed by the organisations, e.g., as in the case of `ibm.com, google.com`.

3.2 DNS Security Extensions (DNSSEC)

Plain DNS requests and responses are not protected and hence expose the DNS resolvers to DNS cache poisoning attacks, whereby altered DNS records, served by malicious entities, redirect the clients to incorrect hosts. Such attacks can be launched by MitM or off-path attackers. For example, a malicious wireless client can tap the communication of other clients and can respond to their DNS requests with maliciously crafted DNS responses, containing a spoofed IP address, e.g., redirecting the clients to a phishing site.

Domain Name System Security Extensions (DNSSEC) standard [RFC4033-RFC4035] was designed to prevent cache poisoning, by providing *data integrity* and *origin authenticity* via cryptographic digital signatures over DNS resource records. The digital signatures enable the receiving resolver, that supports DNSSEC validation, to verify that the data in a DNS response is the same as the data published in the zone file of the target domain.

• New Resource Records (RRs). DNSSEC defines new RRs in order to store signatures and keys which are then used to authenticate the responses. For example, a type RRSIG record contains a signature authenticating an RR-set, i.e., all mappings of a specific type for a certain domain name. `DNSKEY` is the public-key of a zone, which should be used to verify the signatures on resource records for which the zone is authoritative. The signatures are computed using the corresponding private signing key, and are then stored in `RRSIG` RRs. The private signing key should be kept secret and is recommended to be stored offline (to prevent exposure in case a nameserver is compromised).

To be able to verify that the `DNSKEY` is correct, a resolver also obtains a `DS` RR from the parent zone, which contains a hash of the public key of the child; the resolver accepts the `DNSKEY` of the child as authentic if the value in `DNSKEY` is the same as the (hashed) value in the Delegation Signer `DS` record of the parent. Since the `DS` record of the parent is signed, authenticity is guaranteed.

• Trust Anchor and Chain of Trust. In order to validate keys of (possibly) millions of domains, the resolvers should be preconfigured with the public verification key of the root. For validation of keys of target domains, e.g., `foo.bar`, the resolvers need to establish a chain of trust from the root to the keys of the target domain, by following and validating the keys of the intermediate domains.

A sequence of `DNSKEY` and `DS` RRs form a chain of signed data composed of links between each nodes on the path from the target zone to the root. The `DS` RR authenticates the child's `DNSKEY` at the parent. The authentication starts with a set of verified public keys for the DNS root zone which is the trusted third party. This allows the resolver to construct a chain from the root to the target zone's `DNSKEY`. The public key of the parent is used to validate the signature (in `RRSIG`) on the `DS` RR, which contains the public verification key of the child. This way a link is constructed from the child zone to parent zone. The resolver continues this way until a path from the target zone to the root is established. If there is no valid `DS` RR at the parent zone for the child's `DNSKEY`, then the chain of trust is broken and the resolution is not secure.

• DNSSEC Cryptographic Building Blocks. DNSSEC uses a fixed set of cryptographic algorithms and hash functions[6] with the most used being:

[6] `http://www.iana.org/assignments/dns-sec-alg-numbers/dns-sec-alg-numbers.xhtml`

RSA/MD5, RSA/SHA-1, DSA/SHA-1 [RFC4034], and RSA/SHA-256, RSA/SHA-512 [RFC5702]. As we show in Section 5, the most widely supported algorithm is RSA (with different variations). Hence, in the rest of this section we recap RSA and explain two attacks which we will be using in the following sections to factor the vulnerable keys.

3.3 RSA Security and Attacks

In this section we recap the definition of RSA signatures. We then provide two attacks: a shared modulus attack and factorable modulus attack, that we use in this work. The idea behind digital signatures is to allow the receiver to verify that a transmitted message originated from the sender and was not changed by an attacker. DNSSEC uses RSA to ensure validity and authenticity of DNS records.

3.3.1 Definition

The RSA encryption is the most popular scheme used in DNSSEC. The RSA signatures scheme consists of three procedures, key generation, signing and validation, that are defined as follows:

Key Generation. Upon input 1^n, choose two random prime numbers of length $\frac{n}{2}$ and compute $N = p \cdot q$ (and $\gcd(p,q) = 1$). For $N = p \cdot q$ we have $\varphi(N) = (p-1)(q-1)$. Choose e such that $\gcd(e, \varphi(N)) = 1$, and select d such that $e \cdot d \equiv 1 \mod \varphi(N)$. Output $vk = (e,N)$ and $sk = (d,N)$.

Signing. Given sk and m, compute $\sigma = m^d \mod N$, return (m, σ).

Verification. Given $vk = (e,N)$, a message $m \in \{1,...,N-1\}$ and a signature σ, returns 1, if the signature σ is the correct signature on m (i.e., if $\sigma^e \mod N \equiv m$) and returns 0 otherwise. return $c = m^e \mod N$.

3.3.2 Shared Modulus Attack

Lemma 3.1 (Computing Factors of N) *Given RSA public and private keys (e,N) and (d,N) respectively, there exists an efficient algorithm for computing the prime factors of N.*

We provide a proof for a simplified case of small e (e.g., assume $e = 3$) and refer an interested reader to the general case shown in [6].

Proof 3.2 *Given $e \cdot d = 1 \mod \varphi(N)$ and $\varphi(N) = (p-1)(q-1)$, set $e \cdot d - 1 = c(p-1)(q-1)$ for a constant c. For $1 \le e \le 3$ and $1 \le d \le (p-1)(q-1)$ we obtain that $1 \le c \le 3$. To obtain the exact value of c, we can try each of the three values. As a result, we obtain the following equation: $p+q = N+1-\frac{(ed-1)}{c}$. Set $\tau = N+1-\frac{(ed-1)}{c}$ and define the following polynomial: $y(x) = (x-p)(x-q) = x^2 - \tau x + N$. Since τ and N are known and p and q are roots of the polynomial, hence findings the roots also gives the factors of N.* □

Lemma 3.3 (Computing secret key d) *Given RSA public and private keys (e,N) and (d,N) respectively, there exists an efficient algorithm that receives e and factors p and q of N and computes d.*

Proof 3.4 *Given p and q compute $\varphi(N) = (p-1)(q-1)$ then calculate $d = e^{-1} \mod \varphi(N)$.* □

We next consider a setting where a number of participants share the same RSA modulus $N = p \cdot q$, but every participant uses a different public and private keys, namely given ψ participants, each participant i ($i \in \{1,...,\psi\}$) uses (e_i,N) as public key and (d_i,N) as the private key. Then, a malicious party A can recover the private signing key of *any* party i and spoof signatures with private signature key d_i.

Given (e_A,N) and d_A party A can compute p,q, and $\varphi(N)$ and then $d_i = e_i^{-1} \mod \varphi(N)$.

Lemma 3.5 (Private Key Recovery) *Assume (e_i,N) is an RSA public key of party i. There exists an efficient algorithm that given e_i and $\varphi(N)$ calculates d_i.*

First, given (e_i,N) and (d_i,N) party P_i can factor N into p and q. Then, given p and q it can compute $\varphi(N)$, and then d_j for any participant j by computing $d_j = e_j^{-1} \mod \varphi(N)$.

Proof 3.6 *Given p,q compute $\varphi(N) = (p-1)(q-1)$ and find d such that $d = e^{-1} \mod \varphi(N)$.* □

3.3.3 Factorable Modulus Attack

In order to explain this attack, we introduce the Greatest Common Divisor (GCD). Let $a,b > 0$ be integers, the GCD of a and b is the largest integer that divides both a and b.

The importance of GCD with respect to RSA is the following: for every prime number p, every number i between 1 and $p-1$, has a GCD of 1 with p, namely, $\forall i \in \{1,...,p-1\}$, $\gcd(p,i)$ (hence has a multiplicative inverse in modulo p).

If two (or more) different RSA keys share a prime factor, then there are three primes p,q,κ in two keys: $N_1 = p \cdot q$, and $N_2 = p \cdot \kappa$. The reuse of a prime factor allows to calculate all the primes: $\gcd(N_1,N_2) = p$, and obtain q and κ as follows, $q = \frac{N_1}{p}$ and $\kappa = \frac{N_2}{p}$. Then one can compute the secret keys.

The GCD can be computed efficiently using Euclid algorithm. Since RSA moduli are comprised of the product of two prime numbers, they have a common factor with

another number if and only if they share a prime factor. Therefore, knowing any prime factor of the public moduli, results in complete breakage of the key, since the private key can be directly computed using the factors of the modulus and the public exponent.

4 DNSSEC Keys Validation Engine

In this section we present our framework for collecting and processing DNSSEC keys, illustrated in Figure 1. The challenges of our framework are the following: (1) support of dynamic and easy integration of new data sources, (2) perform efficient and fast data collection from the data sources on a daily basis, (3) updates of keys at expiration, (4) enable online validation of keys, domains and registrars.

In the rest of this section we describe the components of our DNSSEC validation engine, including data sources and data collection with DNSSEC keys crawler (that we call DKrawler). We explain how we address the challenges of our framework and briefly review the analysis of the data and its processing into reports, and their presentation on an online web page.

4.1 Data Sources

Our dataset of signed zones uses the following 'crawling seeds':

(1) the root and Top Level Domain (TLD) zone files – we obtained the root and TLD zone files (e.g., for `com, net, org, info`) from the Internet Corporation for Assigned Names and Numbers (ICANN). From the root zonefile we collected all the TLDs, which at the time of our study contained 1301 distinct TLDs (`(http://www.internic.net/domain/root.zone`). We used the Centralized Zone Data Service (CZDS `czds.icann.org`) to obtain the records in the TLD zonefiles.

(2) we scanned the top-1M popular domains according to Alexa `www.alexa.com`. In this work we refer to Alexa domains as SLDs, although many of the Alexa domains also contain third level domains, they typically belong to the same organisation, i.e., the same SLD.

(3) the `sonar DNS project` [37] – performs a daily scan of IPv4 addresses for a range of Internet protocols as well as a full scan of DNS resource records in all the DNS zones.

4.2 DNSSEC Keys Crawler

The first challenge is how to find all the DNSSEC-signed domains in order to collect the cryptographic material from them. To that end, we developed a crawler, which we call DKrawler (DNSSEC Keys crawler) to collect and store DNSSEC-signed domains and their keys. The DKrawler infrastructure is written in Python. The DKrawler periodically (every 24 hours) scans the DNS hierarchy using the seeds that it is configured with. New data sources are continually made available to the public, e.g., recently Cisco Umbrella `s3-us-west-1.amazonaws.com/umbrella-static/index.html` made available a repository of popular domains. To support addition of new data sources we designed our architecture in a modular way. This enables easy inclusion of pluggable modules, which we call "scanners". During each scan key material is collected and inserted into the database. Periodical scan ensures that we cover all the new domains that are added.

The DKrawler collects daily $900K$ signed domains using the data sources in Section 4.1. The number of signed domains slightly varies, as new domains are added while some domains become unavailable. The set of $900K$ signed domains result in $2.1M$ DNSKEY records (i.e., DNSSEC keys) out of which $1.9M$ RSA.

Another challenge is efficient traversal of all the data sources. In particular, this includes two main bottlenecks: (1) the waiting time between sending requests and processing the arriving responses, and (2) crawling the DNS servers asynchronously. Sending requests and processing the responses concurrently would facilitate significant reduction of scan time. To that end, we developed the scanners that operate in a non-blocking mode using "greenlets". Greenlets are a light weight version of threads, that can operate concurrently, and in contrast to multi-processing approach, the greenlets are non-preemptive. This allows sending requests and analysing their responses concurrently.

Greenlets also allow to send multiple DNS requests in parallel and to perform the scan much faster while using significantly less resources (in comparison to the multi-processing approach), reducing the memory requirements from the server and the idle time (when waiting for feedback) while speeding up keys collection by more than 30%. We use up to 1024 greenlets, each scans a domain at a given time point. Hence, we scan a total of 1024 domains in concurrently. The responses are processed and analysed, and the statistics are stored in a MongoDB document database, to facilitate fast insertion and retrieval. The processing requires significant memory and storage resources, for instance, intermediate calculations of the pairwise GCD for more than a million keys.

We developed scripts for producing the following reports: *shared modulus keys*, *shared keys*, *weak keys* and *DNSSEC adoption*. We also measure and provide statistics of key length, and popular cryptographic algorithms. We describe the reports along with the collected statistics in Sections 5 and 6. The reports are produced automati-

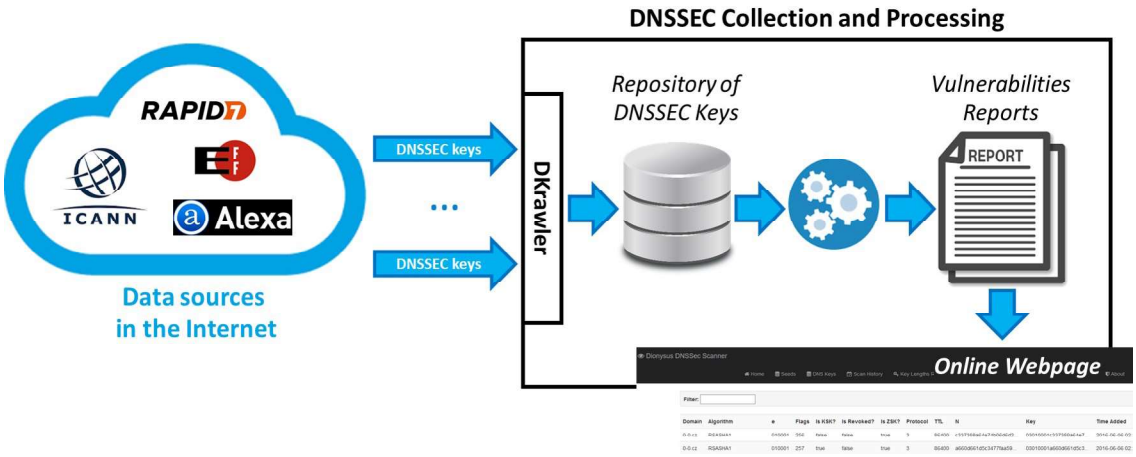


Figure 1: DNSSEC-Keys security framework.

cally after every daily data collection.

The calculations are efficient, on a two core server, computing the shared moduli report takes up to 3.5 minutes, and the weak keys up to 45 minutes, we describe the implementations in Section 6.

The DKrawler (as well as the entire DNSSEC key validation framework) is deployed on the Microsoft Azure cloud, on a VM machine with 2 virtual (Intel Xeon E5-2673 v3 CPU) cores and 14*GB* RAM memory, each clocked at 2.40GHz.

We have developed an online webpage for reporting the daily surveyed domains and the collected keys (more details below).

4.3 Online Reporting of Vulnerable Keys

We provide access to our tool and the database reports via a website at `www.dnssec.sit.fraunhofer.de`. The code is hosted on GitHub (the link to the GitHub project is on the website `www.dnssec.sit.fraunhofer.de`). The web site is based on a lightweight python web framework 'Flask' and MySQL.

The site contains the following tabs: `home` with basic information on scan schedule, seeds and DNS records count; `seeds` lists all the seeds' sources used to collect the data; `DNS Keys` – lists the collected keys and information, such as protocol, TTL and algorithm; `scan history` – previous scans; `key length report` – lists the different key sizes that we collected; `shared moduli report` and `factorable moduli report`.

5 Evaluating DNSSEC Adoption

In this section we provide our measurement of adoption of DNSSEC among the domains in our dataset (Section 4.1): the Top Level Domains (TLDs) and popular domains according to Alexa (these include second and third level domains). We measure the fraction of signed domains, the key sizes and the cryptographic algorithms. As our measurement results indicate, the majority of domains are signed with different variations of RSA algorithms and almost 50% of the deployed RSA keys are shorter than 1500 bit.

5.1 Quantifying Signed Domains

We define DNSSEC-signed domains as those with `DNSKEY` and `RRSIG` records. To check for the fraction of signed domains, we checked for existence of `DNSKEY` and `RRSIG` records in our dataset. Our finding shows that 87.6% of the TLDs are signed, 1.6% of the Alexa domains; we refer to Alexa domains as SLDs, since third level domains are typically within the same SLDs.

In Figures 2 and 3 we plot the results we collected between March and September 2016. In that time interval the number of new TLDs increased by 100 and we observe roughly the same increase in the number of signed TLDs (see Figure 2). This is consistent with the conclusions of our study in the subsequent section, where we show that most registrars and DNS hosting providers perform automated signing of newly registered domains.

We observe a growth in a number of new Second-Level Domains (SLD) domains, however, in contrast to the steady increase in signed TLDs the results indicate a negligible increase in newly signed domains, see plot in Figure 3. The significant and constant growth in the number of signed TLDs indicates that there is an increased awareness to DNSSEC adoption. This increase is also perhaps due to the automated signing adopted by the registrars. In the rest of this work, we study the

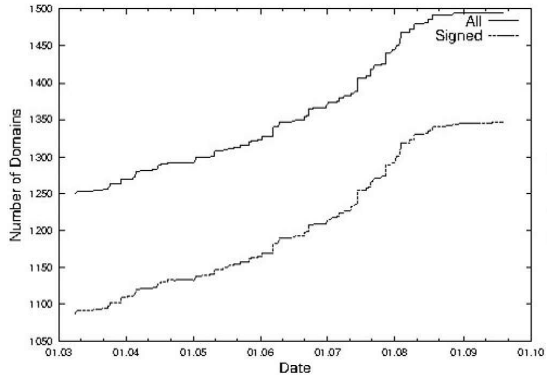


Figure 2: All TLDs and signed TLDs.

signed domains among TLDs and Alexa domains (i.e., SLDs).

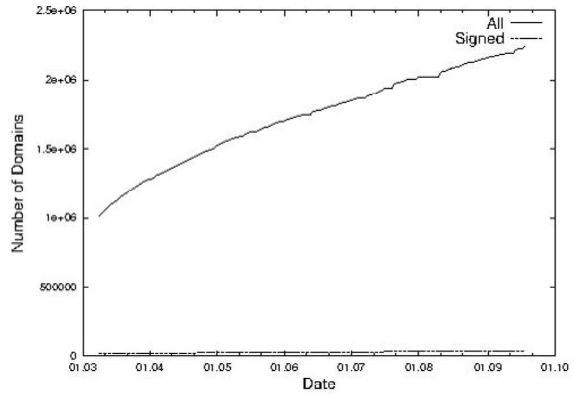


Figure 3: All SLDs and signed SLDs.

5.2 Crypto-Algorithms in Signed Domains

The signed zones can use an arbitrary number of DNSSEC-standardised algorithms (see overview of DNSSEC in Section 3.2). In addition, [RFC4641,RFC6781] list mandatory support for RSA and recommend avoiding large keys (specifying a range of 512-2048 bits for (ZSK) key size and recommending a default value of 1024 bits); in order to avoid fragmentation, communication and computation overhead and other problems with large keys and signatures. In particular, [RFC6781] states "it is estimated that most zones can safely use 1024-bit keys for at least the next ten years.".

Our measurement of DNSSEC adoption among signed domains shows that there is hardly any support for other cryptographic algorithms, e.g., those that produce short signatures, such as ECC, since the motivation to add more overhead to the transmitted data is low. Our results show that RSA, with different digest implementations (SHA1, SHA256, SHA512), dominates among the signed TLDs, and that there is no support for other algorithms among the TLDs, Figure 4. In contrast, there

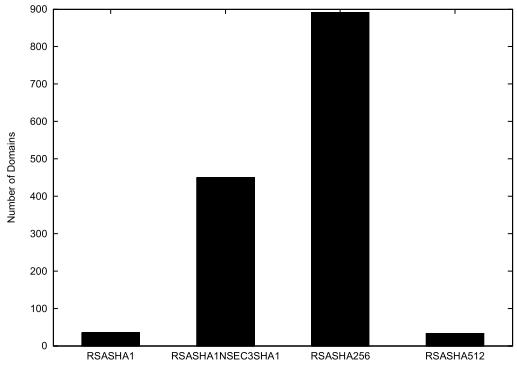


Figure 4: DNSSEC algorithms in signed TLDs.

is some, albeit still limited, attempt to adopt also other cryptographic algorithms, such as DSA and EC in SLDs. These constitute a bit more than 10% of the deployed cryptographic algorithms in DNSSEC, see Figure 5.

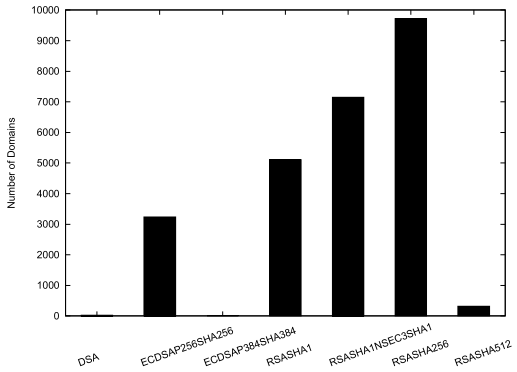


Figure 5: DNSSEC algorithms in signed SLDs.

We measured the key sizes in use by the different variations of RSA algorithms, we plot our results in Figure 6. Almost 1.4M keys are below or equal to 1024 bits, and 10K keys are 512 bits long; [42] showed that factoring 512 bit keys on a cloud is a practical task. For updated statistics on keys and DNSSEC algorithms see our webpage `www.dnssec.sit.fraunhofer.de/key_lengths`.

In a recent work [8] also showed that there are 1% of domains among TLDs and 20% among the SLDs to which it is not possible to establish a chain of trust from the root. The problems include wrong (or missing) `DS`

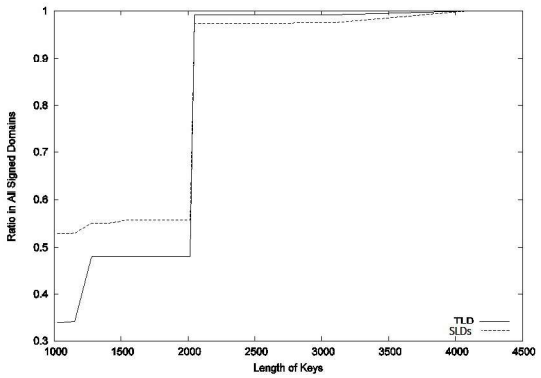


Figure 6: Key sizes in TLDs and SLDs.

records in parent domain, incorrect (or missing) signatures, expired keys, DNSKEY and DS do not match and more. The most common case of broken chain of trust is an existence of DNSKEY but no DS in parent. This may happen when a domain owner wants to enable DNSSEC but his registrar does not support DNSSEC, which is common. Alternately, the same obstacle occurs when the registrar does not support DNSSEC for a TLD under which the domain is registered, e.g., GoDaddy supports DNSSEC only for 10 TLDs. Other common cause is a faulty DS record. This may happen when the domain operator transfers/updates the domain/key or changes the name servers. Broken chain of trust can sometimes be an indication of signed domains in test phase, experiments and deployment in progress.

6 Vulnerable Signed Domains

In this section we describe our measurements of DNSSEC keys generation practices supported by registrars and DNS hosting providers and analysis of vulnerabilities based on the data we collected. We discuss how keys with shared moduli enable recovery of secret signing keys of one domain using the keys of attacker's domain, we show attacks on keys with vulnerable GCD and domains signed with the same keys.

6.1 Shared Keys and Shared Moduli

We differentiate between two cases of vulnerabilities in signed domains: keys with shared moduli and identical keys. In the former case the public validation key e is different, while domains in the latter category have the same N and (e,d). Both practices are vulnerable, in particular, *recovering a key pair for one domain, exposes all other domains signed with the same key or the same modulus, to attacks*.

The attack against domains that are signed using the exact same key pair is straightforward. In particular, once the attacker compromises the key pair, all the domains signed with that key are vulnerable to hijacking. The attacker can generate signatures that will be accepted as valid. Since the records will be signed, the resolvers will assign high trust level to them, allowing to overwrite the previously cached values, [31].

Alternately, it may appear that 'just' reusing the modulus between signed domains does not introduce a vulnerability. We explain that given a key (N,e_1,d_1) the attacker can recover d_2 that belongs to key (N,e_2,d_2). In order to calculate the secret signing keys the attacker needs the knowledge of $\varphi(N)$ (see Section 3.3.2) which it does not have and cannot obtain by querying the DNS information from the domains. To that end, the attacker needs to register and sign a domain under the registrar (or DNS hosting provider), which is used by the (victim) domains whose private key the attacker wishes to recover. Registering domains under most registrars typically costs up to 10$ per year. After the registrar or the hosting provider signs the domain of the attacker, given the key pair which was used to sign its own domain, the attacker computes $\varphi(N)$ and uses the calculation in Section 3.3.2 in order to recover the private signing keys of other domains signed by that registrar (resp. hosting provider). Not all the registrars provide the key pair to the domain owner. In Section 6.1.2 we discuss the countermeasure of keeping the key secret from the domain owner.

Through collection and analysis of keys we found that 700K of the keys were used to sign domains in ways exposing the domains to attacks above. We used `whois` to retrieve registrars' information for domains signed with shared-moduli-keys. We collected the values of the following fields (consider for example `reg-centrum` registrar):

```
registrar:    REG-CENTRUM
address:      CZ
IP location:  Prague
ASN:          AS43614
```

We grouped the domains according to registrars and according to shared-moduli-keys. The resulting groups were similar. Namely, the shared-moduli-keys were typically found within domains under the same registrar. On the other hand, domains signed with shared-moduli-keys are owned by different operators. This suggests that the registrars, generating the DNSSEC keys, are reusing the keys among the domains that they sign.

We registered and signed domains under selected registrars, which had a large fraction of shared-moduli-keys among signed domains. The registrars typically generate the DNSSEC keys on their platforms, then use them to sign the DNS records. The corresponding Delegation

Signer (DS) records are derived from the DNSSEC keys and are updated in the parent domain (typically a TLD).

For example, in one large and popular registrar over $63K$ different domains are signed with two keys ($51,000$ domains are associated with one of the keys and $12,000$ with the other).

We describe our approach for collecting signed domains with shared-moduli-keys.

6.1.1 Finding Domains with Shared Moduli

Given a list of domains, our tool first collects all the DNSKEY records for signed domains. Then, the tool iterates over all the domains and creates a mapping between each modulus and domains that use that modulus. The tool checks if the public verification keys of the signed domains are different or the same. The domains are signed with a single key when the public e are identical.

```
collect-keys(List):
  For each domain D in List:
key = dns.request(DNSKEY)
    store-in-DB(key)

map-modulus(List):
  d = new dictionary()
  For each domain D in List:
   modulus = D.modulus
 d[modulus].append(D)
```

At the completion, `d` contains a mapping between the different moduli and domains that use that moduli. To extract a list of domains that share a modulus we iterate over `d` and collect domains with length greater than 1. In Figure 7 we plot the results of our calculations of domains with shared-moduli-key over a period of March-September 2016. The graph shows that the fraction of signed domains with shared-moduli-keys is stable over time, and slightly increases as new domains are signed. The majority of moduli and keys are shared among few domains (approximately 10,000). Alternately, few modulus values are shared among a large fraction of signed domains (almost 14%). These moduli are associated with signed domains operated by large registrars.

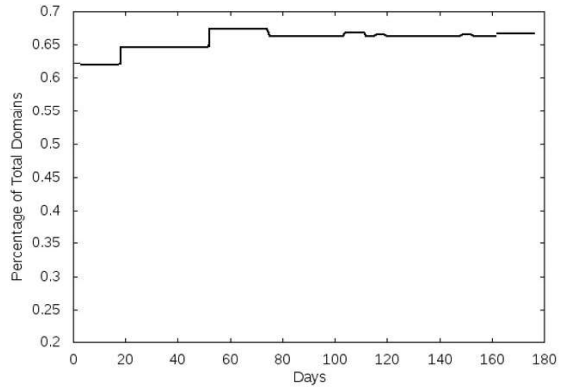


Figure 7: March-September 2016 measurements of fraction of domains with shared moduli (count per day).

6.1.2 Hiding Secret Keys as a Countermeasure

If the registrars, supporting DNSSEC, allow domain owners to obtain the keying material pertaining to their signed domains, the attackers can use this to attack domains signed with the same keys or signed with keys that share modulus values. In what follows we consider whether withholding the crpytographic material would be a viable solution against the vulnerabilities.

The RSA modulus is advertised as part of the public key, and is needed for verification of DNSSEC signatures. According to the Lemmas in Section 3.3.2, given a secret key of one domain, an attacker can recover a secret signing key of another domain (that is signed under the same RSA modulus). Hence it may appear that to counter the vulnerabilities due to a shared RSA modulus, a registrar should hide the secret signing keys from the owners of signed domains.

As we next argue, a naive countermeasure of hiding the RSA secret keys from the domain owner does not constitute a good defence and, moreover, is often not practical. First and foremost, using a single key for singing multiple domains, while keeping it secret from the domain owners, leads to an insecure practice. In particular, *a compromise of the secret key, e.g., via compromise of a registrar, would lead to compromise of the security of all the domains which were signed using that key*. Compromises of registrars are common, e.g., [48].

Hiding the keys could also limit the domain owners to using only the nameservers of one specific registrar - the one that performed the DNSSEC signing. This stands in contrast to the best practices of DNS - which recommend maintaining servers under at least two domains. For instance, consider a scenario where a domain owner registers a domain, which gets signed by the registrar, and decides to use the nameservers provided by that registrar. When the operator adds its own nameserver, and configures as master, he would need a secret signing key to sign the zone file also on its own nameserver.

Our study of ICANN accredited registrars shows that none of the registrars that support DNSSEC, enable it for all TLDs. For instance, GoDaddy enables DNSSEC only for ten TLDs. As a result, if the domain owner wishes to follow best practices and place its domain under two TLDs, it will often not be able to, if it does not have the secret signing key used by one of the registrars.

6.2 Even Moduli & Keys w/Shared Primes

Distinct moduli that share a prime factor will result in public keys that appear distinct but whose private keys are efficiently computable by calculating the greatest common divisor (GCD). For calculation of GCD of every pair of keys we followed the approach in [22] and used the *fast pairwise GCD* quasilinear-time algorithm for factoring a collection of integers into coprimes; we compiled and used the source code (`https://factorable.net/resources.html`) provided by [22].

The code implements an efficient (quasilinear time) algorithm to compute the GCD of every pair of integers.

After calculating group-GCD on all the DNSKEY records, we found that out of factorable moduli 16 RSA moduli were even. Re-querying the nameservers multiple times for these moduli returned the same results over the period of our measurement. Among the possible causes for this could be either a faulty key generation or bit-errors, e.g., due to heat, on the machines generating and storing them. This situation also indicates no DNS servers (nor the platforms operated by the registrars and DNS hosting providers) validate the correctness of the generated DNSKEY records.

The keys with even RSA moduli belonged to domains hosted or registered by known registrars, such as Network Solutions, GoDaddy, OnlineNic. In Figure 8 we plot our measurements of factorable RSA keys, collected over a period of March-September 2016.

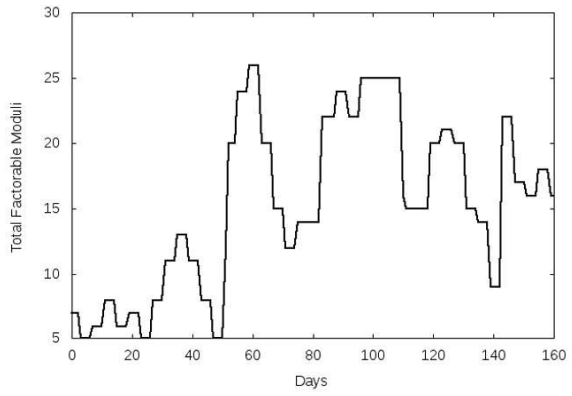


Figure 8: March-September 2016 measurements of fraction of domains with factorable moduli (count per day) as a function of the total number of keys.

6.3 Poisoning Signed DNS Records

We next describe steps for poisoning caches of DNSSEC-validating victim resolvers with spoofed DNS records for signed domains. Assume the attacker at IP `6.6.6.6` wishes to hijack emails sent by clients on network `1.2.3.0/24` to an email server in domain `vic.tim`, and assume that the domain `vic.tim` is signed with DNSSEC. Further, assume that `vic.tim` is hosted with registrar Reg that uses the same modulus N for all its keys. In step (1) the attacker registers and signs a domain `att.ack` with Reg. (2) the attacker uses the secret signing key and the modulus (as in Section 6.1) to calculate the secret signing key for `vic.tim`. (3) the attacker creates a valid signature over a spoofed record pointing an email server at `vic.tim` to IP 6.6.6.6. (4) the attacker performs cache poisoning attack injecting a signed record `mail.vic.tim A 6.6.6.6`. Notice that DNSSEC signed DNS records are assigned the highest trust level by the caches, and hence overwrite any other previously cached value associated with the record.

6.4 Discussion

What are the causes for the phenomenon of shared keys and moduli in DNSSEC keys? The answer is perhaps related to the recent ICANN regulation, [40], requiring that the registrars and DNS hosting providers support DNSSEC for all domains they host or register. On the one hand, the registrars and DNS hosting providers are required to offer and support DNSSEC, and to automate the signing and keys generation processes for the customers. A notable example for this is Binero, which performs the keys generation and signing by itself. Automation of DNSSEC certainly facilitates faster deployment thereof among unsigned domains. The regulation is important as it quickly increased the fraction of domains adopting DNSSEC. On the other hand, automating keys generation requires integrating the necessary software, support on the web interface, access to sources of randomness and using suitable hardware and processes. The registrars and DNS hosting providers may be tempted to save on randomness, and to reuse keys among multiple domains that they sign. As we showed, this practice is vulnerable and can potentially lead to an *illusion of security*, while rendering the *signed* domains exposed to key recovery attacks, and their clients (i.e., DNSSEC validating resolvers) to DNS cache poisoning.

Automating adoption of cryptography is an important goal, and a notable effort undertaken by ICANN [40], however, to facilitate it, the registrars and DNS hosting providers should be equipped with suitable procedures to provide secure services to their customers.

7 Countermeasures and Defences

The community spent a considerable effort to design and operate tools for validation of DNSSEC. For instance Verisign provides `http://dnssec-debugger.verisignlabs.com/`, Sandia

National Liboratories operate `http://dnsviz.net/`, the French operator AFNIC operate the Zonemaster `http://zonemaster.net/`. Upon input a domain name, these tools perform analyses of the configuration of the domain and deployment of DNSSEC, e.g., correctness of the signatures or ability to establish a path to the trust anchor. Since these tools operate on each domain in isolation they cannot detect keys or moduli sharing.

Another set of tools validate correctness of zonefile signing. In particular, they verify that the keys of a target domain sign the DNS records, and that all the required DNSSEC records (such as NSEC or NSEC3) are present in the zonefile. These tools enable domain owners to identify a broken chain of trust.

Unfortunately, these tools do not detect *vulnerable* keys, such as those with a shared or factorable moduli. Tools for detection of vulnerabilities in cryptographic keys would enable the registrars and DNS hosting providers to identify the problems in the keys that they generate, hence preventing the faulty keys from being stored and served by the DNS servers.

Our recommendation is to prohibit sharing of DNSSEC keys across multiple domains. Such a regulation could be overseen by The Internet Corporation for Assigned Names and Numbers ICANN (`www.icann.org`) or Internet Assigned Numbers Authority IANA (`iana.org/`). However, even if put forth, such policies cannot be validated without tools. In what follows we describe two aspects of our DNSSEC-Keys validation engine: one allows online validation of generated keys, the other produces lists of vulnerable domains signed with shared keys. Both validations are integrated into our framework (Section 4). We explain the defences and refer an interested reader to the online webpage of our framework (`www.dnssec.sit.fraunhofer.de/`).

Online Keys Validation Service

Our DNSSEC-keys validation engine (Section 4) supports validation of keys and of signed domains. Given a key in an input, we automatically perform the following checks: (1) the RSA modulus is not even; (2) the new modulus does not collide with the stored modulus values of the scanned domains; (3) the new modulus does not share prime factors with the stored moduli values of the scanned domains. Performing such a validation requires comparing each newly generated key to the keys of (multiple other) signed domains. By comparing a given key to all other keys we verify that the key does not share a modulus with other domains and that it is not composed of prime factors shared with keys in other domains.

This service can be used by domain owners as well as by registrars to ensure security of generated keys.

Listing Vulnerable Domains

We keep a dynamic list of vulnerable domains signed with same keys or domains signed with keys sharing modulus values. This enables customers to evaluate the security offered by the different registrars and hosting services.

Such lists can also be used by firewalls to trigger alerts when clients access vulnerable domains, or by browsers (via a simple browser extension) to alert web clients about a potential security risk, since the domain is not signed.

8 Conclusions

While SSL and TLS received a significant attention from the research and operational communities, and prior work measured vulnerabilities in crypto-algorithms in the wild, the adoption of cryptography used by DNSSEC-signed domains requires more attention. In this work we perform an Internet-wide study of keys in signed DNSSEC domains. To that end, we first measure adoption of DNSSEC and collect a dataset of RSA signed domains. We then validate the security of the keys in our collected set of signed domains. Our results indicate that multiple domains are signed with shared keys. The vulnerabilities stem from poor key generation practices used by registrars and DNS hosting operators.

We developed a DNSSEC keys validation engine, to periodically collect, analyse and process the DNSSEC keys used by TLDs and other popular domains. We set up a website for online reporting of the analyses over the collected data, as a service to the community. Our online service supports validation of keys, enabling clients to identify vulnerabilities in newly generated keys, before the keys are used to sign the zone files and published in online repositories.

9 Acknowledgements

We thank Gal Beniamini and Matan Ben Yosef for developing the tool for collection and processing of keys and Stephane Bortzmeyer, Amit Klein and Roy Arends for their helpful comments on our manuscript. We are also grateful to the anonymous referees for their thoughtful feedback on our work. The research reported in this paper has been supported in part by the German Federal Ministry of Education and Research (BMBF) and by the Hessian Ministry of Science and the Arts within CRISP (www.crisp-da.de/). We are grateful to Microsoft Azure Research Award, which provided hosting for our research infrastructure.

References

[1] D. Anderson. Splinternet behind the great firewall of china. *Queue*, 10(11):40, 2012.

[2] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, et al. Drown: Breaking tls using sslv2.

[3] D. J. Bernstein. Nsec3 Walker. http://dnscurve.org/nsec3walker.html, 2011.

[4] D. J. Bernstein, Y.-A. Chang, C.-M. Cheng, L.-P. Chou, N. Heninger, T. Lange, and N. Van Someren. Factoring rsa keys from certified smart cards: Coppersmith in the wild. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 341–360. Springer, 2013.

[5] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudorandom bits. *SIAM journal on Computing*, 13(4):850–864, 1984.

[6] D. Boneh et al. Twenty years of attacks on the rsa cryptosystem. *Notices of the AMS*, 46(2):203–213, 1999.

[7] B. Chor and O. Goldreich. Unbiased bits from sources of weak randomness and probabilistic communication complexity. *SIAM Journal on Computing*, 17(2):230–261, 1988.

[8] T. Dai, H. Shulman, and M. Waidner. Dnssec misconfigurations in popular domains. In *International Conference on Cryptology and Network Security*, pages 651–660. Springer, 2016.

[9] D. Davis, R. Ihaka, and P. Fenstermacher. Cryptographic randomness from air turbulence in disk drives. In *Advances in CryptologyCRYPTO94*, pages 114–120. Springer, 1994.

[10] C. Deccio, J. Sedayao, K. Kant, and P. Mohapatra. A case for comprehensive dnssec monitoring and analysis tools. *Proceedings of SATIN*, 2011.

[11] C. Deccio, J. Sedayao, K. Kant, and P. Mohapatra. Quantifying and improving dnssec availability. In *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*, pages 1–7. IEEE, 2011.

[12] L. Dorrendorf, Z. Gutterman, and B. Pinkas. Cryptanalysis of the windows random number generator. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 476–485. ACM, 2007.

[13] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, et al. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.

[14] K. Fukuda, S. Sato, and T. Mitamura. A technique for counting dnssec validators. In *INFOCOM, 2013 Proceedings IEEE*, pages 80–84. IEEE, 2013.

[15] H. Gao, V. Yegneswaran, Y. Chen, P. Porras, S. Ghosh, J. Jiang, and H. Duan. An empirical reexamination of global dns behavior. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 267–278. ACM, 2013.

[16] I. Goldberg and D. Wagner. Randomness and the netscape browser. *Dr Dobb's Journal-Software Tools for the Professional Programmer*, 21(1):66–71, 1996.

[17] S. Goldberg, M. Naor, D. Papadopoulos, L. Reyzin, S. Vasant, and A. Ziv. Nsec5: Provably preventing dnssec zone enumeration. *IACR Cryptology ePrint Archive*, 2014:582, 2014.

[18] S. Goldberg, M. Naor, D. Papadopoulos, L. Reyzin, S. Vasant, and A. Ziv. Stretching nsec3 to the limit: Efficient zone enumeration attacks on nsec3 variants. 2015.

[19] O. Gudmundsson and S. D. Crocker. Observing DNSSEC Validation in the Wild. In *SATIN*, March 2011.

[20] P. Gutmann. Software generation of random numbers for cryptographic purposes. In *Proceedings of the 1998 Usenix Security Symposium*, pages 243–257, 1998.

[21] Z. Gutterman, B. Pinkas, and T. Reinman. Analysis of the linux random number generator. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[22] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 205–220, 2012.

[23] A. Herzberg and H. Shulman. Security of patched DNS. In *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, pages 271–288, 2012.

[24] A. Herzberg and H. Shulman. Fragmentation Considered Poisonous: or one-domain-to-rule-them-all.org. In *IEEE CNS 2013. The Conference on Communications and Network Security, Washington, D.C., U.S.* IEEE, 2013.

[25] A. Herzberg and H. Shulman. Socket Overloading for Fun and Cache Poisoning. In C. N. P. Jr., editor, *ACM Annual Computer Security Applications Conference (ACM ACSAC), New Orleans, Louisiana, U.S.*, December 2013.

[26] A. Herzberg and H. Shulman. Vulnerable delegation of DNS resolution. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, pages 219–236, 2013.

[27] A. Herzberg and H. Shulman. Retrofitting Security into Network Protocols: The Case of DNSSEC. *Internet Computing, IEEE*, 18(1):66–71, 2014.

[28] M. Hu. Taxonomy of the snowden disclosures. *Wash & Lee L. Rev.*, 72:1679–1989, 2015.

[29] D. Kaminsky. It's the End of the Cache As We Know It. In *Black Hat conference*, August 2008. `http://www.blackhat.com/presentations/bh-jp-08/bh-jp-08-Kaminsky/BlackHat-Japan-08-Kaminsky-DNS08-BlackOps.pdf`.

[30] A. Klein. BIND 9 DNS cache poisoning. Report, Trusteer, Ltd., 3 Hayetzira Street, Ramat Gan 52521, Israel, 2007.

[31] A. Klein, H. Shulman, and M. Waidner. Internet-Wide Study of DNS Cache Injections. In *INFOCOM*, 2017.

[32] P. Levis. The collateral damage of internet censorship by dns injection. *ACM SIGCOMM Computer Communication Review*, 42(3), 2012.

[33] W. Lian, E. Rescorla, H. Shacham, and S. Savage. Measuring the Practical Impact of DNSSEC Deployment. In *Proceedings of USENIX Security*, 2013.

[34] D. Liu, S. Hao, and H. Wang. All your DNS records point to us: Understanding the security threats of dangling DNS records. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1414–1425, 2016.

[35] C. Meyer and J. Schwenk. SoK: Lessons learned from SSL/TLS attacks. In *International Workshop on Information Security Applications*, pages 189–209. Springer, 2013.

[36] K. Schomp, T. Callahan, M. Rabinovich, and M. Allman. Assessing dns vulnerability to record injection. In *Passive and Active Measurement*, pages 214–223. Springer, 2014.

[37] security research project by Rapid7. Project Sonar, 2013-2016.

[38] H. Shulman and M. Waidner. Fragmentation Considered Leaking: Port Inference for DNS Poisoning. In *Applied Cryptography and Network Security (ACNS), Lausanne, Switzerland*. Springer, 2014.

[39] H. Shulman and M. Waidner. Towards security of internet naming infrastructure. In *European Symposium on Research in Computer Security*, pages 3–22. Springer, 2015.

[40] I. Society. ICANNs 2013 RAA Requires Domain Name Registrars To Support DNSSEC, 2013.

[41] J. Stewart. Dns cache poisoning–the next generation, 2003.

[42] L. Valenta, S. Cohney, A. Liao, J. Fried, S. Bodduluri, and N. Heninger. Factoring as a service.

[43] R. van Rijswijk-Deij, M. Jonker, A. Sperotto, and A. Pras. A high-performance, scalable infrastructure for large-scale active dns measurements. *IEEE Journal on Selected Areas in Communications*, 34(6):1877–1888, 2016.

[44] M. Wander, C. Boelmann, L. Schwittmann, and T. Weis. Measurement of globally visible dns injection. *Access, IEEE*, 2:526–536, 2014.

[45] M. Wander, L. Schwittmann, C. Boelmann, and T. Weis. Gpu-based nsec3 hash breaking. In *Network Computing and Applications (NCA), 2014 IEEE 13th International Symposium on*, pages 137–144. IEEE, 2014.

[46] F. Weimer. DSA-1571-1 openssl - Predictable Random Number Generator, 2008.

[47] H. Yang, E. Osterweil, D. Massey, S. Lu, and L. Zhang. Deploying cryptography in internet-scale systems: A case study on dnssec. *Dependable and Secure Computing, IEEE Transactions on*, 8(5):656–669, 2011.

[48] Z. Zorz. Lenovo.com Hijacking Made Possible by Compromise of Webnic Registrar, 2015.

Enhancing Security and Privacy of Tor's Ecosystem by using Trusted Execution Environments

Seongmin Kim, Juhyeng Han, Jaehyung Ha, Taesoo Kim*, Dongsu Han
KAIST Georgia Tech*

Abstract

With Tor being a popular anonymity network, many attacks have been proposed to break its anonymity or leak information of a private communication on Tor. However, guaranteeing complete privacy in the face of an adversary on Tor is especially difficult because Tor relays are under complete control of world-wide volunteers. Currently, one can gain private information, such as circuit identifiers and hidden service identifiers, by running Tor relays and can even modify their behaviors with malicious intent.

This paper presents a practical approach to effectively enhancing the security and privacy of Tor by utilizing Intel SGX, a commodity trusted execution environment. We present a design and implementation of Tor, called SGX-Tor, that prevents code modification and limits the information exposed to untrusted parties. We demonstrate that our approach is practical and effectively reduces the power of an adversary to a traditional network-level adversary. Finally, SGX-Tor incurs moderate performance overhead; the end-to-end latency and throughput overheads for HTTP connections are 3.9% and 11.9%, respectively.

1 Introduction

Tor [35] is a popular anonymity network that provides anonymity for Internet users, currently serving 1.8 million users on a daily basis [13]. Tor provides sender anonymity through multi-hop onion routing/encryption as well as responder anonymity using "rendezvous points" that allow the operation of *hidden services*. It is a volunteer-based network in which world-wide volunteers donate their computation and network resources to run open-source Tor software. At the time of this writing, Tor consists of 10,000 relays, with some relay nodes even known to be run by a variety of law enforcement agencies around the world [5, 15]. However, it is not without limitations.

Fundamentally, Tor is vulnerable when an attacker controls a large fraction of relays; anonymity (or privacy) can be broken if all relays in a circuit are compromised because Tor relays can identify the circuit using its identifiers. To prevent malicious relays from entering the system, Tor exercises a careful admission and vetting process in admitting new relays and actively monitors their operation. At the same time, to make traffic analysis more difficult, Tor relies on having a large number of relays and tries to keep a diverse set of relays spread out world-wide [33, 34], which helps to decrease the chance of selecting two or more relays controlled by an adversary. However, having a large network and keeping all relays "clean" are fundamentally at odds in a volunteer-based network. This is exemplified by the fact that, by design, Tor relays are not trusted; in operation they are carefully admitted and their behaviors are examined by a centralized entity [27, 35].

Even having control over a relatively small number of Tor relays still gives significant advantages to attackers. For example, a malicious adversary can change the behavior by running a modified version of Tor, compromise keys, and/or have access to other internal information, such as the circuit identifier, header, and hidden service identifiers. In fact, many *low-resource attacks* (i.e., attacks that do not require taking control of a large fraction of the network) heavily rely on adversaries acquiring internal information or being able to modify the behavior of Tor relays. These low-resource attacks utilize a combination of multiple factors, such as being able to demultiplex circuits, modify the behavior, and access internal data structures. For example, harvesting hidden service identifiers [27] requires access to a relay's internal state, a sniper attack [43] requires sending false `SENDME` cells, and tagging attacks [60] require access to header information. Selective packet drop [27, 43] or circuit closure [28], used by many attacks, also requires being able to demultiplex circuits with circuit identifiers.

This paper aims to address the current limitations of Tor and practically raise the bar for Tor adversaries by using Intel SGX, a commodity trusted execution environment (TEE) available on the latest Skylake and Kaby Lake

microarchitectures. We ask ourselves three fundamental questions: *(1) What is the security implication of applying TEE on Tor? (2) What is its performance overhead?* and *(3) Is it deployment viable in the current Tor network?*

To this end, we design and implement SGX-Tor, which runs on real SGX hardware. We show that it can effectively reduce the power of Tor adversaries to that of a network-level adversary that cannot see the internal state of Tor components. Specifically, we protect private Tor operation, such as TLS decryption and circuit demultiplexing, from adversaries, so that only the TLS-encrypted byte stream is exposed to them, unlike the original vanilla Tor. We further argue that this has far-reaching implications on the trust model and operation of Tor:

- **Trust model:** Currently, Tor relays are semi-trusted. While they are monitored and vetted during admission and operation, their behaviors are not fully verified. In fact, many attacks are discovered and reported after the fact [20, 43, 60, 68]. With SGX-Tor, behaviors are verified through attestation, and private information is securely contained without being exposed to untrusted parties. This simplifies the vetting and monitoring process, allowing Tor to grow its capacity more easily. This will provide a stronger foundation for Tor's privacy (anonymity).
- **Operation and deployment:** SGX-Tor has significant implications in Tor operation. First, because we can both prevent and detect code modification and forging false information, many attacks can be prevented. Second, because SGX-Tor controls the information exposed to the external world, it helps operational privacy. For example, we can ensure that the consensus document, which lists Tor relays and their states [35], does not leave the secure container (i.e., enclave). This effectively turn all relays into bridge relays, a set of relays that not publicly listed [18]. Finally, SGX-Tor can be easily deployed because it uses commodity CPUs and can even be incrementally deployed.

In summary, we make the following contributions:

1. We analyze the assumptions and components used in existing attacks on Tor and discuss how the use of Intel SGX nullifies them to disable the attacks.
2. We present the first design and implementation of Tor that run on real SGX hardware.
3. We demonstrate that SGX-Tor limits the power of Tor adversaries to that of a network-level adversary.
4. We characterize the performance of Tor-SGX through extensive micro- and macro-benchmarks.

Organization: § 2 provides background on Intel SGX and Tor. § 3 describes our approach and the attacks SGX-Tor can defend against. § 4 and § 5 provide the system design and implementation, which we evaluate in § 6. § 7 discusses remaining issues and concerns. § 8 presents related work, and § 9 concludes our work.

2 Background

This section provides key features of Intel SGX and an overview of the Tor anonymity network.

Intel SGX: Intel SGX provides isolated execution by putting and executing the code and data of an application inside a secure container called an *enclave*. It protects sensitive code/data from other applications and privileged system software such as the operating system (OS), hypervisor, and firmware. The memory content of the enclave is stored inside a specialized memory region called Enclave Page Cache (EPC). The EPC region of memory is encrypted within the Memory Encryption Engine (MEE) of the CPU and is hardware access controlled to prevent snooping or tampering with the enclave page content.

Intel SGX instructions consist of privileged instructions and user-level instructions. Privileged instructions are used for loading application code, data, and stack into an enclave. When the enclave is loaded with appropriate memory content, the processor generates the identity of the enclave (i.e., SHA-256 digest of enclave pages) and verifies the integrity of the program by checking the identity that contained a signed certificate (EINIT token) of the program. If the verification succeeds, the CPU enters the enclave mode and the program within the enclave starts to execute from a specified entry point. User-level instructions are used after the program loads.

SGX also provides remote attestation and sealing functions. Remote attestation allows us to verify whether the target program is executed inside an enclave without any modification on a remote SGX platform [24]. Finally, sealing allows us to store enclave data securely in a non-volatile memory by encrypting the content using a `SEAL_KEY`, provisioned by SGX CPU [24]. Unseal restores the content back into the enclave. Intel white papers [39, 40, 53] describe the specifications in detail.

Tor network: The Tor network is a low-latency anonymity network based on onion routing [35]. Tor consists of three components: clients (Tor proxies), directory servers, and relays. Suppose that Alice uses Tor proxy to communicate with Bob through the Tor network. By default, Alice's proxy sets up 3-hop (entry, middle, exit) onion-encrypted circuit to ensure that any single Tor component cannot identify both Alice and Bob (e.g., entry relay knows the source is Alice, but does not know who Alice is talking to). Directory servers are trusted nodes that provide signed directories, called the consensus document. They consist of nine computers run by different trusted individuals and vote hourly on which relays should be part of the network. Relays are provided by volunteers who donate the hosting platform and network bandwidth.

Relays maintain a TLS connection to other relays and clients and transmit data in a fixed-size unit, called a cell.

Each relay maintains a long-term identity key and a short-term onion key. The identity key is used to sign the router descriptor and TLS certificates, and the onion key is used for the onion routing. The directory server also uses an identity key for TLS communication and a signing key for signing the consensus document.

Tor also provides receiver anonymity. It allows Bob to run a hidden service behind a Tor proxy and serve content without revealing his address. To publish a service, Bob's Tor proxy chooses a relay that will serve as an introduction point (IP) and builds a circuit to his IP. It then creates a hidden service descriptor containing its identifier (ID), IP, and public key and publishes the information in the Tor network. The descriptor is stored in a distributed hash table called a hidden service directory. Using the descriptor obtained from the directory, Alice establishes a circuit to its IP and specifies a rendezvous point (RP) for Bob. The IP then relays this information to Bob. Finally, the RP forwards communication between Alice and Bob.

3 Approach Overview

This section describes our assumptions and threat model, presents high-level benefits of applying SGX to Tor, and analyzes how SGX-Tor prevents many attacks.

3.1 Scope

In this paper, we focus on attacks and information leakage that target Tor components. Because Tor is a volunteer-based network, an attacker can easily add malicious relays and/or compromise existing relays. Subversion of directory authorities seriously damages the Tor network, which needs to be protected more carefully. We also consider attacks and information leakage that require colluding relays. Obtaining control over Tor nodes is relatively easier than manipulating the underlying network [66] or having wide network visibility [29, 44, 56]. We follow Tor's standard attack model [35] and do not address attacks that leverage plain text communication between client and server and network-level adversaries (e.g., traffic analysis and correlation attacks [44]).

Threat model: We take a look at how Tor's security model can be improved with SGX. Instead of trusting the application and the system software that hosts Tor relays, SGX-Tor users only trust the underlying SGX hardware. We assume an adversary who may modify or extract information from Tor relays. Following the threat model of SGX, we also assume an adversary can compromise hardware components such as memory and I/O devices except for the CPU package itself [53]. In addition, any software components, including the privileged software (e.g., operating system, hypervisor, and BIOS), can be inspected and controlled by an attacker [53]. DoS attacks are outside the scope of this paper since malicious system software or hardware can simply deny the service (e.g., halt or reboot). Also, side channel attacks, such as cache timing attacks on SGX, are also outside the scope. Both assumptions are consistent with the threat model of Intel SGX [53] and prior work on SGX [26, 61]. Finally, software techniques for defending against attacks that exploit bugs [62, 64] (e.g., buffer overflow) in in-enclave Tor software is out-of-scope.

3.2 SGX-Tor Approach and its Benefits

Main approach: First, our approach is to enclose all private operation and security-sensitive information inside the enclave to make sure that it is not exposed to untrusted parties. We make sure that private or potentially security-sensitive information, such as identity keys and Tor protocol headers, does not leave the enclave by design, relying on the security guarantees of the SGX hardware. This ensures that volunteers do not gain extra information, such as being able to demultiplex circuits, by running a Tor node other than being able to direct encrypted Tor traffic.

Second, we prevent modification of Tor components by relying on remote attestation. When Tor relays are initialized, their integrity is verified by the directory servers. Thus, directory servers ensure that relays are unmodified. Directory servers also perform mutual attestation. We also extend this to attest SGX-enabled Tor proxies (run by client or hidden server) for stronger security properties. Unless otherwise noted, we primarily consider a network in which all Tor relays and directory servers are SGX-enabled. We explicitly consider incremental deployment in §4.2. In the following, we summarize the key benefits of the SGX-Tor design and its security implications.

Improved trust model: Currently, Tor relays are semi-trusted in practice. Some potentially malicious behaviors are monitored by the directory server, and others are prevented by design. However, this does not prevent all malicious behaviors. The fundamental problem is that it is very difficult to explicitly spell out what users must trust in practice. This, in turn, introduces difficulties to the security analysis of Tor. By providing a clear trust model by leveraging the properties of SGX, SGX-Tor allows us to reason about the security properties more easily.

Defense against low resource attacks: To demultiplex circuits, low resource attacks often require node manipulation and internal information that is obtained by running Tor relays. Examples include inflating node bandwidth [27], sending false signals [43], injecting a signal using cell headers [21, 27], and packet spinning attack [58]. SGX-Tor prevents modifications to the code and thus disables these attacks (see §3.3).

Leakage prevention of sensitive information: Directory servers and Tor relays use private keys for sign-

ing, generating certificates, and communicating each other through TLS. Directory servers are under constant threats [32]. If directory authorities are subverted, attackers can manipulate the consensus document to admit or to direct more traffic to malicious relays. Multiple directory authorities have been compromised in practice [32]. This caused all Tor relays to update their software (e.g., directory server information and keys). Relays also contain important information, such as identity keys, circuit identifiers, logs, and hidden service identifiers. By design, SGX-Tor ensures that data structures contained inside the enclave are not accessible to untrusted components, including the system software.

Operational privacy: The consensus document distributed by the director servers lists Tor relays. However, keeping the information public has consequences. It is misused by ISPs and attackers to block Tor [11], to infer whether the relay is being used as a guard or exit [28, 57], and to infer whether Tor is being used [56]. The information also has been used in hidden server location attacks [27, 57]. As a counter-measure, Tor maintains bridge relays. Currently, users can obtain a small number of bridge addresses manually. When all Tor nodes, including Tor proxies, are SGX-enabled, one can keep the list of all relays private by sending the consensus document securely between the directory and user enclaves to enhance the privacy of the Tor network.

3.3 Attacks Thwarted by SGX-Tor

Attacks on Tor typically use a combination of multiple attack vectors. To demonstrate the benefit of SGX-Tor, we analyze existing attacks on Tor and provide a security analysis for SGX-Tor. First, we show attacks that require node modification and how SGX-Tor defeats them.

A bandwidth inflation [25, 27, 57] attack exploits the fact that clients choose Tor relays proportional to the bandwidth advertised in the consensus. This provides malicious relays an incentive to artificially inflate their bandwidths to attract more clients [27]. Bandwidth inflation has been one of the key enablers in low resource attacks [25]. When a relay is first introduced in the network, it reports its bandwidth to the directory servers, allowing the relay to falsely report its bandwidth. To prevent the relays from cheating, directory servers scan for the relay's bandwidth. However, the bandwidth probing incurs pure overhead. It can also be evaded by throttling other streams to make scanners misjudge [27] the bandwidth of relays. Leveraging this, Biryukov et al. [27] inflated the bandwidth report more than 10 times. SGX-Tor simplifies bandwidth reports because of the enhanced trust model. A relay just needs to report the sum of bandwidth that it uses to serve Tor traffic. Because it can be trusted, we do not need an external bandwidth scanner. Note SGX-Tor also defeats replay attacks that might be mounted by an untrusted OS (e.g., duplicate messages) by generating a nonce and keeping it within the enclave during the TLS connection between the relay and directory server.

Controlling hidden service directories [27]: Tor relays that have an HSDir flag set serve as hidden service directories by forming a distributed hash table to which hidden services publish their descriptors. To use a hidden service, clients must fetch the hidden service descriptor that contains the descriptor ID, the list of introduction points, and the hidden service's public key. Biryukov et al. [27] demonstrated an attack in which the attacker can control access to any hidden services. First, malicious relays become hidden service directories for the target hidden services. This amounts to generating a public key that falls into an interval in which the hidden service descriptor ID belongs. After this, malicious hidden service directories can see the requests for the target hidden service descriptors, which they can drop to deny the service. SGX-Tor prevents this because 1) untrusted components in the relay do not see the descriptor; and 2) relay behaviors cannot be altered.

A tagging attack is a type of traffic confirmation attack that changes the characteristics of a circuit at one end (e.g., exit relay) for it to be recognized by the other (e.g, entry guard). This is used to effectively de-anonymize communication between two parties. These attacks require modification of relays. For example, a cell counting attack [27, 50], replay attack [60], and relay early traffic confirmation attack [21] send more command cells, duplicate existing cells, or adjust how many cells are sent over the network at a time to create a distinct pattern. SGX-Tor prevents them because these attacks require relay modification. Note that tagging has been used to de-anonymize hidden services. Biryukov et. al. [27] modified the rendezvous point to send 50 `PADDING` followed by a `DESTROY` cell. Clients use the rendezvous point and the attacker can reliably determine if the hidden service uses its relay as the entry by counting the number of cells. If a pattern is found, the previous node to the entry is marked as the hidden service.

Consensus manipulation in directory server: By taking over directory servers, attackers can manipulate the consensus by accessing the memory content of directory servers. This allows them to admit malicious relays, cast a tie-breaking vote, or steal keys [35]. Especially, the admission of malicious relays is very dangerous; it increases the possibility of various low-resource attacks using malicious Tor relays. During the vetting process, the directory authority creates a "status vote," which contains the information of relays such as its liveness and bandwidth information. The authorities then collect the voting result and generate a consensus document. If more than half of the authorities are manipulated, they can publish the consensus document that contains many malicious

relays [35]. SGX-Tor not only prevents attackers from accessing the content by placing the information inside the enclave, but also detects modified directory servers.

Second, some attacks do not require node modification, but break the privacy by leveraging a relay's internal information. SGX-Tor prevents this by limiting the information available to the attackers.

Collection of hidden service descriptors [27, 51]: This attack collects all hidden service descriptors by deploying a large number of relays that serve as hidden service directories (HSDir). Obtaining service descriptors is easy because one can just dump the relay's memory content. It is shown that with careful placement of HSDirs in the distributed hash table, 1,200 relays is are enough to harvest the entire list [27], which is used to launch other attacks, such as opportunistically de-anonymizing hidden services. The use of SGX-Tor prevents this, as all potentially security-sensitive information, including the hidden service descriptor, is stored only inside the enclave.

Demultiplexing and finger-printing: Tor multiplexes multiple circuits in a single TLS connection and multiple streams (e.g., TCP flows) in a single circuit. Many attacks rely on being able to identify circuits and streams. For example, cell counting attacks [27, 50] and circuit and website finger-printing attacks [47] take advantage of the relay's ability to identify and count the number of cells in a circuit. Traffic and timing analysis used by Overlier et. al. [57] leverages circuit-level information to strengthen the attack. In a vanilla Tor circuit, demultiplexing is trivial because each relay decrypts the TLS connection. In contrast, SGX-Tor hides circuit-level information, including identifiers, from the rest of the world. Note that this means running Tor relay does not give any more information than being a network-level adversary that observes traffic. This makes traffic/finger-printing analysis attacks much more difficult because now an adversary must rely on an additional layer of inference (e.g., timing analysis) for circuit demultiplexing. This forces adversaries to take more time and resources to successfully mount an attack and increase the false positive rates for finger-print attacks, especially in a heavily multiplexed environment [17, 22, 34, 59]. Thus, it enhances the privacy of Tor users.

Bad apple attack [48]: Making circuit identification non-trivial also raises the bar for the bad apple attack. In Tor, multiple TCP streams from a user share the same circuit because it improves efficiency and anonymity [35]. However, this means that even if one TCP stream's source address is revealed, the source address of all TCP streams within a circuit is revealed. The attack takes advantage of this and uses "insecure" applications to de-anonymize secure applications within the same circuit [48]. With SGX-Tor, the attack is not as straightforward because an SGX-enabled exit node that observes many TCP streams cannot easily associate the streams with their circuit. Even when the node is observing all packets, circuit association is difficult on a highly multiplexed exit node (e.g., even if the predecessor is the same for two packets, they may belong to different circuits). A more involved traffic analysis and long running TCP sessions may be required.

Finally, clients (Tor proxies) have also been used to launch attacks. We discuss how SGX-Tor can protect Tor against existing attacks with SGX-enabled Tor proxies.

A sniper attack [43] is a destructive denial-of-service attack that disables Tor relays by making them to use an arbitrarily large amount of memory [43]. A malicious client sends a `SENDME` signal through the circuit without reading any data from it. `SENDME` causes the exit relay to send more data, which exhausts memory at the entry, eventually causing it to be terminated by the OS. This attack requires Tor proxy (client) modification, which can be prevented when the proxy uses SGX. When using SGX-enabled proxies, directory servers or entry guards can verify their integrity. When there is a mix of non-SGX and SGX clients, an effective counter-measure would be to kill circuits [19]. when an entry guard is short of memory, but it can deprioritize circuits to the SGX-enabled proxies when looking for victims because they can be trusted.

Malicious circuit creation: Congestion [37] and traffic analysis attacks [54, 55] use throughput information as a side channel to break the anonymity of Tor (e.g., de-anonymize relays offering a hidden service, identify guards or relays used by a flow). These attacks commonly modify clients to create 1-hop circuits or circuits with a loop to inject measurement traffic to target relays. An SGX-enabled Tor proxy can prevent this by enforcing a minimum hop for circuits (e.g., 3) and disallowing loops when a proxy creates a circuit. Without creating a loop, throughput finger-printing is made much more difficult, less accurate, and more expensive.

Hiding consensus document: As explained in §3.2, SGX-enabled clients and directory servers can keep the list of relays private by enclosing the consensus information inside the enclave to enhance operational privacy.

4 Design

SGX-Tor ensures, by design, the confidentiality of security-sensitive data structures and the integrity of Tor nodes. In addition to the direct benefits of applying SGX, SGX-Tor is designed to achieve the following goals:

Trustworthy interface between the enclave and the privilege software: Although Tor must rely on system software (e.g., system calls) for operation, the interface between the enclave and operating system (OS) must not be trusted. A malicious or curious operating system (or even firmware) can compromise applications running on

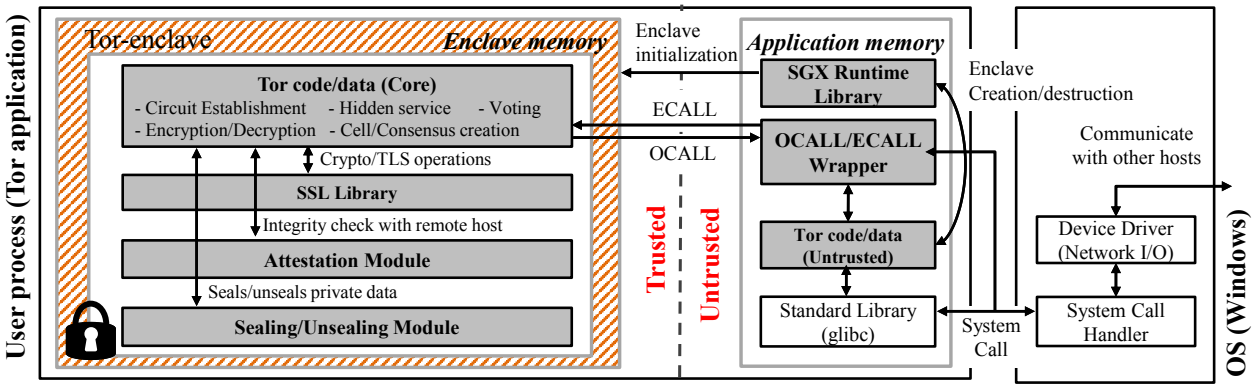


Figure 1: The architecture of SGX-Tor. Gray-colored boxes indicate modified or newly added components from the original Tor. The address space of the Tor application is divided into: *enclave memory* and *application memory*. The enclave communicates only with untrusted software through a well-defined interface.

a secure enclave by carefully manipulating interface between them (e.g., return values in system calls [30]). To reduce such an attack surface, we define a narrow interface between the untrusted and trusted components of SGX-Tor, making the interface favorable to verification or formal proof [26]. For example, SGX-Tor relies on minimal system support for networking, threading, and memory allocation and performs sanity-checking for input/output arguments inside the enclave when requesting services to untrusted system software.

Reducing performance overhead: Utilizing SGX causes inevitable performance degradation for two main reasons: 1) context switches occurring when entering and leaving the enclave mode require TLB flushes and memory operations for saving/restoring registers, and 2) memory accesses (not cache accesses) require additional encryption or decryption by a Memory Encryption Engine (MEE). In addition, the small EPC region (e.g., 128 MB in Intel Skylake [6]) can limit the active working set and thus requires further management of enclave memory that incurs additional performance overhead. Since SGX-Tor protects all security-sensitive data structures and operations within the enclave, large data structures (e.g., list of router descriptors whose size is 10 MB) easily deplete the EPC capacity, in which case the kernel evicts EPC pages through SGX paging instructions (e.g., EWB and ELDB/U) [53], leading to performance degradation. To reduce the overhead, we minimize copying already encrypted data to EPC, such as encrypted packets, and explicitly stage out from EPC large data structures that are used infrequently while ensuring their confidentiality with sealing (see §5).

Deployability and Compatibility: We make practical recommendations to achieve compatibility with existing Tor. Our design facilitates incremental deployment of SGX-enabled Tor components. Note that some features such as remote attestation must have SGX-enabled directory servers, and some properties are only achieved when all components are SGX-enabled. In this paper, we discuss potential issues and benefits of incremental deployment of the SGX-enabled Tor ecosystem.

4.1 SGX-Tor: Architecture

Figure 1 shows the overall architecture of SGX-Tor, shared by all components. The memory region is divided into two parts: the hardware-protected enclave memory region and the unprotected application memory region. Tor-enclave, staged inside an enclave, contains the core components, such as directory authorities, onion routers, and client proxy, which are protected. The untrusted components in the application memory implement an interface to system calls and non-private operations, such as command line and configuration file parsing.

Tor-enclave also contains essential libraries for Tor applications. The SSL library handles TLS communication and the cryptographic operations (e.g., key creation) required for onion routing. The remote attestation module provides APIs to verify the integrity of other Tor programs running on the remote side. The sealing module is used when sensitive information such as private keys and consensus documents must be stored as a file for persistence. SGX-Tor uses the sealing API to encrypt private keys and consensus documents with the seal key provided by the SGX hardware. The enclosed file is only decrypted within the enclave through the unsealing API.

The unprotected application code provides support for Tor-enclave without handling any private information. It handles public data, such as RSA public keys, published certificates, and router finger-prints. Note that key pairs and certificates are generated in Tor-enclave. The SGX runtime library provides an interface to create or destroy an enclave. The untrusted part and the Tor-enclave run as a single process, communicating through a narrow and well-defined interface. A function that enters the enclave is called an `ECALL`, and a function that leaves the enclave is called an `OCALL` as defined in the Intel SGX

SDK [6]. Table 4 (in Appendix A) lists major E/OCALL interfaces. We use ECALLs to bootstrap Tor-enclave, while OCALLs are used to request services to the system software. The OCALL wrapper of SGX-Tor passes the request (e.g., send() system call) and arguments from the enclave (e.g., buffer and its length) to the system software and sends the results back to the enclave. Tor-enclave relies on the following system services:

- Network I/O (socket creation, send/recv packets)
- Threading and signal management for event handlir
- Error handling
- Memory mapping for file I/O

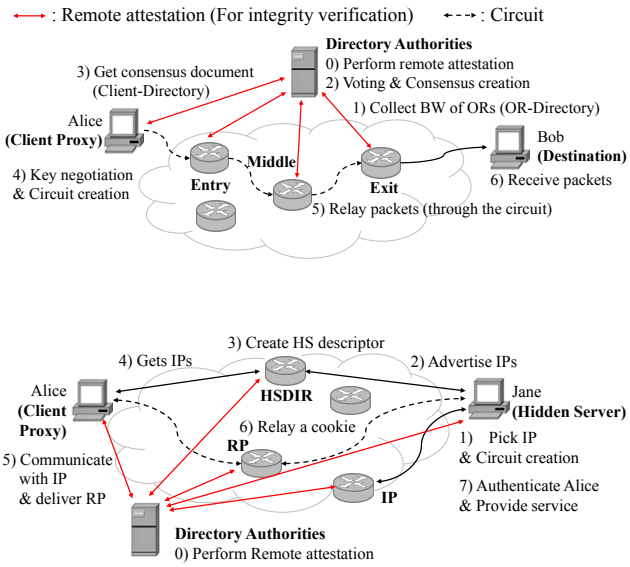
Note, we rely on the I/O and resource allocation se vices provided by the system software. In addition providing the narrow interface, we harden the interfac because the OCALL interface and its wrapper are untruste we validate the parameters and return a value of OCAI For example, we perform sanity-checking for paramete of the OCALL interface to defend against attacks (e.g buffer overflow) from the untrusted code/data by utilizir the Intel SGX SDK. For every system call used by Tc we leverage this feature to validate the pointer variables provided by the untrusted OS by putting additional arguments (if needed) that specify the size of the input/output buffer [1].

4.2 SGX-Tor Components and Features

Figure 2 (a) describes the Tor components for providing sender anonymity and (b) illustrates the scenario for a hidden service (i.e., responder anonymity). SGX-Tor applies SGX to every component, including client proxy, directory authorities, onion routers, and hidden services. This section describes how each component is changed in SGX-Tor. We first present the design of four common features shared by all components, followed by the individual Tor components shown in Figure 2.

Initialization (common): All Tor components except client proxy create key pairs, a certificate, and the finger-print at initialization. For this, Tor provides a tor-gencert tool that creates RSA keys and certificate for directory authorities and Tor relays. Directory servers create private keys for signing and verifying votes and consensus documents. A Tor relay creates an onion key to encrypt and decrypt the payload for onion routing. Both directory and relay have an identity key pair to sign TLS certificates and consensus document/router descriptor. The original Tor saves the key pairs as a file, which can be leaked once the privilege software is compromised.

[1]For example, ocall_sgx_select(), an OCALL for select(int nfds, ..., struct timeval *timeout) has an additional parameter "**int tv_size**" to specify the buffer size of **"timeout"** (See Appendix). The value is filled in inside the enclave and the sanity-checking routine provided by the SDK inspects the input/output buffer within the enclave.



(b) Onion routing for the hidden service

Figure 2: Overview of SGX-Tor in action. The remote attestation verifies the integrity of each other.

SGX-Tor protects the cryptographic operations and seals private keys before storing them in a file.

TLS communication (common): When Tor forwards packets between relays within a circuit, it uses TLS to encrypt the application-level Tor protocol, such as circuit ID, command, cell length, and payload. Currently, this information is visible to relays and system software that hosts the relays. Thus, security-sensitive information, including session key and related operations for establishing TLS connection (e.g., handshaking, encryption, and decryption), must be protected from the untrusted software. In SGX-Tor, the TLS communication is executed within the enclave, leaving the payload of the Tor protocol protected. The system software only handles the network I/O of packets that are encrypted within the enclave.

Sealing of private information (common): When a Tor application terminates, it stores the cached consensus documents and private/public key pairs to a second storage space for later use. To securely perform such an operation, SGX-Tor utilizes the sealing function. When the Tor-enclave must store private information in a file, it encrypts the data using a seal key provided by SGX hardware. The stored data can be loaded and decrypted when the same program requests an unseal within an enclave. Based on the sealing/unsealing interface in the SGX SDK, we develop a high-level API to store the important data of the directory authorities and client proxies. The sealed data is never leaked, unless the CPU package is compromised.

Supporting incremental deployment (common): So far, we explained the system, assuming that all parts

are SGX-enabled. However, we also support interoperability; e.g., it is possible to establish a circuit with an entry-guard that runs SGX-Tor while the middle and exit relays run the original Tor. We add configuration options in the Tor configuration file (`torrc`) to enable remote attestation. The `EnableRemoteAttest` option is set by directory authorities to indicate whether it supports remote attestation. It also has `RelaySGXOnly` and `ClientSGXOnly` options to only admit relays and clients that pass attestation. The relays and clients can set the `RemoteAttestServer` option to request attestation to the directory. For the SGX-Tor client proxy we add an option to get the list of validated relays from the SGX-enabled directory server. Without these options, SGX-Tor behaves like an ordinary Tor without attestation.

Directory authority: The directory authority manages a list of Tor relays from which the client proxy selects relays. The consensus document, containing the states of Tor relays, is generated by directory servers through voting that occurs every hour. The voting result (i.e., consensus document) is signed by the directory authority to ensure authenticity and integrity. SGX-Tor creates a consensus document and performs voting inside the enclave. For example, data structures for keeping the relay's bandwidth information (`networkstatus_t`), voter list, and voting results are securely contained inside the enclave.

Onion router (relay): Tor relays perform encryption/decryption of the cell content. Relays periodically rotate the private onion keys used for onion routing. SGX-Tor encloses such operations inside the enclave so that security-sensitive information, such as circuit identifiers, cannot be manipulated by an attacker. Because TLS communication is also performed inside the enclave, untrusted components cannot observe Tor commands, unlike in the original Tor. Finally, bandwidth measurement, stored in the `routerinfo_t` data structure, is done securely by calculating the sum of bandwidth inside the enclave so that it cannot be inflated or falsely reported.

Client proxy: The client's circuit establishment and key negotiation process with Tor relays are securely executed inside the enclave. Also, the consensus document and the list of relays are enclosed within the enclave, unlike in the original Tor, where they are transmitted using TCP unencrypted. We modified it to use TLS inside the enclave so that keys and consensus documents are not exposed to untrusted components. We also disallow clients creating a loop in a circuit. For hidden services, SGX-Tor securely manages relevant data structures, such as the hidden service descriptor, address of rendezvous point, and circuit identifier for hidden services to prevent any unintended information leakage.

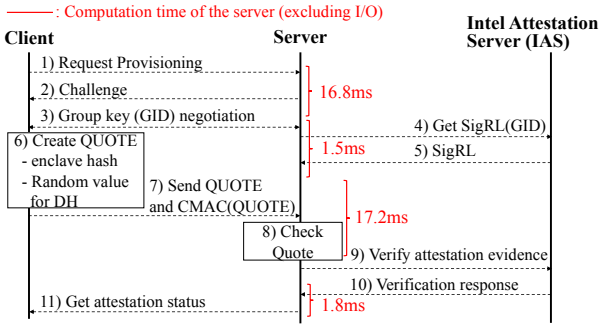


Figure 3: Remote attestation procedure. QUOTE contains the hash of the target enclave in the client.

4.3 Remote Attestation

SGX-Tor uses remote attestation to detect and remove modified nodes from the network. Currently, the same Tor binary serves as the directory authority, Tor relay, and client proxy. Only their configurations are different. This means that every Tor component has the same measurement.

Figure 3 shows the attestation procedure between a client in which an enclave program runs and a remote server that is verifying its integrity. Intel provides Intel Attestation Server (IAS) [41], whose role is similar to a certificate authority, to aid the process; it provisions to the remote server a public key used to authenticate the attestation report. It also issues an endorsement certificate for each SGX processor's attestation key to ensure that the key is stored within the tamper-resistant SGX CPU [31]. During the remote attestation, the remote server checks the `QUOTE` data structure that contains the hash value (code/data pages) to verify the integrity of the client. The remote server then signs the `QUOTE` using the Enhanced Privacy ID (EPID) group key and forwards it to the IAS. The IAS verifies the signature and `QUOTE` and sends the attestation report to the remote server. Here, the EPID group key provides anonymity and also supports unlinkability [41]. Finally, the server verifies the report signature and sends the attestation status to the client.

We use remote attestation for a) integrity verification of relays during onion router (or relay) admission, b) mutual attestation between authorities, and c) sending the list of relays to the trustworthy client. SGX-Tor provides high-level APIs for each remote attestation cases.

Integrity verification of relays (Dir-to-Relay): When a relay contacts directory authorities to register itself in the Tor network, it requests remote attestation (taking the client role in Figure 3) to the directory authorities (server role). The directory server admits only "clean" relays and filters out suspicious ones that run modified Tor programs. Non-SGX relays will also fail to pass the attestation.

Mutual attestation of directories (Dir-to-Dir): The directory authorities mutually perform remote attestation to detect modified servers. A modified directory might try

Component	Lines of code (LoC)	% Changed
Tor application	138,919 lines of C/C++	2.5% (3,508)
EDL	157 lines of code	100% (157)
OCALL wrapper	3,836 lines of C++	100% (3,836)
Attestation/Sealing	568 lines of C/C++	100% (568)
OpenSSL	147,076 lines of C	1.0% (1,509)
libevent	21,318 lines of C	7.0% (1,500)
zlibc	8,122 lines of C	-
Scripts	262 lines of python	-
Total	321,470 lines of code	3.4% (11,078)

Table 1: Lines of code for SGX-Tor software.

to admit specific Tor relays (possibly malicious), launch tie-breaking attacks to interrupt consensus establishment, or refuse to admit benign relays [35]. However, because malicious directory servers will fail to pass the attestation with SGX-Tor, it helps the Tor community to take action quickly (e.g., by launching a new reliable authority). When the codes related to the relay admission policy or algorithm of the directory server are patched, a new measurement can be distributed to check its validity through remote attestation. Note that the existing admission control mechanisms for Tor relays can still be used.

Trustworthy client (Client-to-Dir): To detect modified client proxies, the directory authority attests clients and transmits only the consensus document when they pass attestation. This filters out modified clients that might perform an abnormal circuit establishment such as creating a loop [37, 54, 55] and also keeps the consensus document confidential. Therefore, only benign SGX-enabled Tor clients obtain the consensus document, which contains the list of relays verified by directory authorities. In summary, if all relays and clients are SGX-enabled, we can 1) keep the list of relays private and 2) block malicious clients.

5 Implementation

We developed SGX-Tor for Windows by using the SGX SDK provided by Intel [6]. In total, it consists of 321K lines of code and approximately 11K lines of code are modified for SGX-Tor, as broken down in detail in Table 1. As part of this effort, we ported OpenSSL-1.1.0 [8], zlib-1.2.8 [16], and libevent-2.0.22 [7] inside the enclave. Note that the porting effort is non-trivial; for one example, OpenSSL libraries store the generated keys to files, but to securely export them to non-enclave code, we have to carefully modify them to perform sealing operations instead of using naive file I/Os. Furthermore, because enclave programs cannot directly issue system calls, we implemented shims for necessary system calls with an `OCALL` interface. However, to minimize the TCB size, we ported only required glibc functions, such as `sscanf()` and `htons()`, instead of embedding the entire library. As a result, our TCB becomes dramatically reduced compared to other SGX systems such as Graphene [67] or Haven [26] (more than 200 MB) that implements a whole library OS to support SGX applications; SGX-Tor results in 3.8 times smaller TCB compared to Graphene (320 K vs. 1,228 K LoC). The source code is available at `https://github.com/kaist-ina/SGX-Tor`.

Managing enclave memory: To work with the limited EPC memory, SGX-Tor seals and stores large data structures *outside* of the enclave. If required, it explicitly loads and unseals the encrypted data into the enclave. For example, `cached-descriptors` that contain the information of reachable relays (e.g., finger-print, certificate, and measured bandwidth), are around 10 MB for each, which is too big to always keep inside the enclave. Unlike the original Tor, which uses memory-mapped I/Os to access these data, SGX-Tor loads and unseals them into the EPC only when it has to update the list of relays, which essentially trades extra computation for more usable EPC memory. Similarly, certain system calls such as `recv()` are implemented to save the enclave memory; they get a pointer pointing to the data (e.g., encrypted packets) outside the enclave instead of copying them to the enclave memory.

Sealing and unsealing API: We implemented sealing and unsealing API to substitute file I/O operations for private keys. SGX-Tor uses C++ STL map to store generated private keys in the enclave memory. The key of the map is the name of the private key, and the value of the map is a structure that contains the contents and length of a private key. When SGX-Tor needs to read a generated key, it finds the key contents by the key name through the map. The application side of SGX-Tor can request the private keys using sealing API to store it in the file system. SGX-Tor uses sealing before sending the keys outside the enclave. In reverse, the application side of SGX-Tor also can request to load the sealed private keys using unsealing API. SGX-Tor decrypts sealed key by unsealing it and stores it in the map. These sealing and unsealing mechanisms are easily usable because they are implemented as macros.

Securely obtaining entropy and time: The vanilla OpenSSL obtains entropy from the untrusted underlying system through system calls, like `getpid()` and `time()`, that make the enclave code vulnerable to Iago attacks [30, 42]; for example, a manipulated time clock can compromise the logic for certification checking (e.g., expiration or revocation). To prevent such attacks, we obtain entropy directly from the trustworthy sources: randomness from the `rdrand` instruction (via `sgx_read_rand`) and time clocks from the trusted platform service enclave (PSE) (via `sgx_get_trusted_time`) [6][pp. 88-92, 171-172].

Data structure	Tor	Network-level adversary	SGX-Tor (Component)
TCP/IP header	V	V	V
TLS encrypted bytestream	V	V	V
Cell	V	**N**	**N** (R)
Circuit ID	V	**N**	**N** (R)
Voting result	V	**N**	**N** (D)
Consensus document	V	**N**	**N** (D/R/C)
Hidden service descriptor	V	**N**	**N** (H)
List of relays	V	**N**	**N** (C)
Private keys	V	**N**	**N** (D/R/C)

Table 2: Information visible to adversaries who run SGX-Tor and original Tor and network-level adversaries. "V" denotes visible; "N" denotes non-visible. Component "D" denotes a directory authority, "R" relay, "C" client, and "H" hidden service directory.

6 Evaluation

We evaluate SGX-Tor by answering three questions:

- What types of Tor attacks can be mitigated?
- What is the performance overhead of running SGX-Tor? How much does each component of SGX-Tor contribute to the performance degradation?
- How compatible is SGX-Tor with the current Tor network? How easy is SGX-Tor adopted?

Experimental setting: We set up two evaluation environments for SGX-Tor: 1) by admitting SGX-Tor onion router in the real Tor network and 2) by constructing a private Tor network where all components, including directories and client proxies, run SGX-Tor. We used nine SGX machines (Intel Core i7-6700 3.4GHz and Intel Xeon CPU E3-1240 3.5GHz). The private Tor network consists of a client proxy, five relays, and three directory servers. Note that directory servers also work as relays. We extend the work of Chutney [23] to configure and orchestrate our private SGX-Tor network.

6.1 Security Analysis

Table 2 summarizes security- and privacy-sensitive data structures that are available to three types of adversaries: 1) an adversary who controls relays running original Tor, 2) an adversary who controls the platform running SGX-Tor, 3) and a network-level adversary. V marks the visible information to an adversary, whereas N denotes non-visible ones. An adversary who controls the vanilla Tor can access a great deal of sensitive information, attracting more adversaries to run malicious Tor relays. In contrast, an attacker who even controls the platform running SGX-Tor cannot gain any information other than observing the traffic, just like a network-level adversary. This indicates that the power of Tor adversaries is reduced to that of network-level adversaries with SGX-Tor. Among the attacks in §3.3, we choose three well-known classes of attacks considering their reproducibility and severity. We replicate these attacks (and their key attack vectors) in a lab environment and evaluate if SGX-Tor correctly mitigates them.

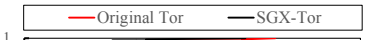
Bandwidth inflation: To demonstrate this attack, we modify the Tor code to advertise inflated bandwidth of a relay to directory servers. The directory server performs bandwidth scanning to check whether a relay actually serves the bandwidth advertised by itself [10]. During the scanning, the directory server creates a 2-hop circuit, including the target relay, and downloads the file from particular hosts to estimate the bandwidth of the relays. If a malicious relay is selected as non-exit, it can directly see which connection is originated from the directory server [27]. By throttling other traffic, the compromised relay inflates the measured bandwidth and gets a fast flag, which is given to a high bandwidth relay indicating that they are suitable for high-bandwidth circuits, in the consensus document [25, 27, 57]. However, with SGX-Tor, modifying the Tor code is not fundamentally possible due to the measurement mismatch during the attestation.

Circuit demultiplexing: Being able to decrypt cell Tor headers and demultiplex circuits and streams in relays is a common attack vector exploited in cell counting attacks [27, 50], traffic analysis [57], website finger-printing attacks [47], bad apple attacks [48], replay attacks [60], relay early attacks [21], and controlling access to hidden services [27]. With a modified relay, we were able to dump Tor commands, circuit IDs, and stream IDs; count cells per stream [27, 50]; duplicate cells [60]; and even selectively drop particular circuits and streams [28]. However, with SGX-Tor, the modified relay failed to be admitted due to attestation failure. With the attested SGX-Tor relay, it is not possible to dump the EPC memory outside the enclave unless the code inside the enclave is compromised due to an exploitable bug (e.g., buffer overflow). Even inferring the cell boundary was not trivial, let alone observing decrypted cell headers.

Malicious circuit creation: By modifying the original Tor code, we successfully establish a 3-hop loop circuit. Creating a loop can be an attack vector for traffic analysis [54, 55] and congestion attack [37] with a long loop. In SGX-Tor, it is not possible to introduce loops because the directory authority can verify the integrity of the client proxies. Therefore, the modified Tor client fails to manipulate a circuit as it intended.

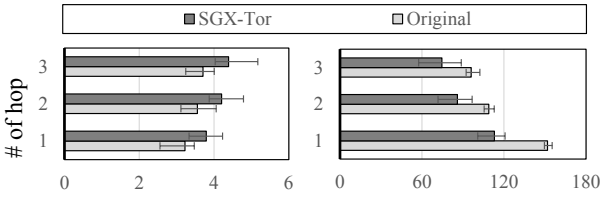
6.2 Performance Evaluation

End-to-end performance: To quantify the effect on performance in a wide-area network, we configure a private Tor network that consists of an entry guard and exit relay located in East Asia and the U.S. East, respectively. For SGX-Tor, every Tor component, including client proxy, except the destination server runs SGX, except for mid-



Type	Time-to-fisrt-byte (ms)
Original (baseline)	2.05
SGX-Tor (overhead)	2.19 (6.8%)

Table 3: Overhead of TLS co



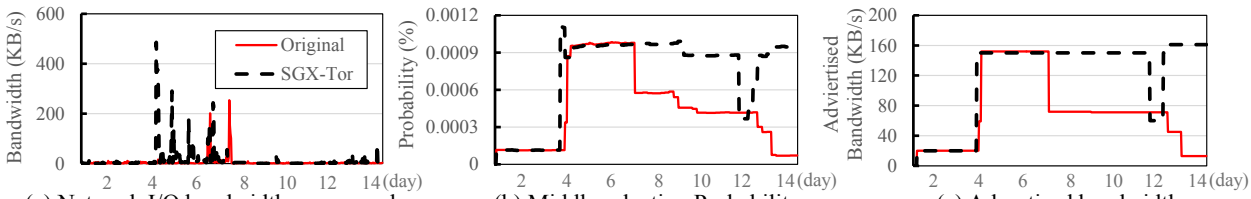
(a) Time-to-first-byte (ms) (b) Throughput (Mbps)

Figure 10: Overhead of onion encryption.

dle relays. To diversify the middle
use Amazon EC2 [2] U.S. East, U.S
instances. Figure 4 shows the CDF
downloading a file (10 MB) via a H
The results are based on the average o
exhibits 11.9% lower throughput (3.11 Mbps) for HTTP and 14.1% (2.95 Mbps) lower throughput for HTTPS. Figure 5 shows the CDF of time-to-first-byte (latency) for HTTP transfer. SGX-Tor (525ms) only gives 3.9% additional delays compared to the original Tor (505ms). We also evaluate the web latency when a client connects to a website through Tor. Figure 6 shows the distribution of the web page loading time for Alexa Top 50 websites [1]. We measure the time from the initiation of a request until an onload event is delivered in the Firefox browser. Similar to time-to-first-byte, SGX-Tor gives 7.4% additional latency on average. We note that our SGX SDK allows compilation only in debug mode, since it requires an approved developer key provided by Intel to run an enclave in release mode. Thus, we used debug mode for all Tor performance measurements.

Hidden service: To quantify the overhead of running a hidden service with an SGX-Tor proxy, we run one on the real Tor network. At the client side, we use a Tor browser [12] that automatically picks a rendezvous point. For each measurement, we relaunch a Tor browser to establish a new circuit. We perform 100 measurements that transfer a 10 MB file from an HTTP file server running as a hidden service. Figure 7 shows the distribution of the throughput for SGX-Tor and the original Tor. The hidden service using SGX-Tor gives only a 3.3% performance degradation from 1.35 to 1.30 Mbps on average. We see a smaller gap because the performance is more network bottlenecked as packets from/to a hidden service traverse two 3-hop circuits and only the hidden service uses SGX.

Overhead of TLS and onion encryption: To quantify the overhead in a more CPU-bound setting, we create a private Tor network in which all components are connected locally at 1Gbps through a single switch. We measure the overhead of SGX-Tor starting from a single TLS connection without any onion routing and increase the number of onion hops from one to three. Table 3 shows the time-to-first-byte and throughput of TLS communication without onion routing. The result shows that SGX-Tor has 9.99% (716 to 651 Mbps) of performance degradation and has 6.39% additional latency (2.05 to 2.19 ms). Figure 10 shows the time-to-first-byte and throughput by increasing the number of onion hops for downloading a 10 MB file from the HTTP server. As the hop is increased, SGX-Tor has 17.7%, 18.3%, and 18.4% additional latency and the performance is degraded by 25.6%, 21.1%, 22.7%, respectively. Figure 8 shows the end-to-end client performance in the private Tor net-



(a) Network I/O bandwidth per second (b) Middle selection Probability (c) Advertised bandwidth

Figure 11: Compatibility test of SGX-Tor and original Tor while running as a middle relay. Both Tor relays started at the same time and acquire "Fast" and "Stable" flags during the evaluation [3].

work. The vanilla Tor achieves 106 Mbps for HTTP and 101 Mbps for HTTPS transfers, while SGX-Tor gives 85 Mbps and 77 Mbps respectively, resulting in a throughput degradation of 24.7% and 31.2%, respectively (for 10 MB).

Remote attestation: We quantify the latency and computation overhead of remote attestation. To emulate a Tor network running in a wide area setting, we introduce latency between the SGX-Tor directory and relays in our private Tor network to mimic that of the real network. The round-trip time between East Asia (where most of our SGX servers are) and nine directory authorities in Tor falls between 144ms (longclaw [3] in the U.S.) and 313ms (maatuska [3] in Sweden). The round-trip time between our directory and IAS was 310ms. We execute 30 runs for each directory while introducing the latency of a real Tor network. Figure 9 shows the CDF of the remote attestation latency when the directory authority verifies Tor relays. The average attestation latency of the entire process in Figure 3 is 3.71s.

SGX-Tor relays and clients request the remote attestation to the directory server once during bootstrapping. This means that this is a one-time bootstrapping cost and is relatively short compared to the client bootstrapping time, which takes 22.9s on average on a broadband link to download a list of relays [49]. We quantify the computational cost of remote attestation for the directory authority for verifying client proxies. For attestation, directory servers calculate the AES-CMAC of group ID for checking EPID group and ECDSA signature generation for QUOTE verification. On our i7 machine, the computation takes 37.3ms in total if the QUOTE verification succeeds, as annotated in Figure 3. When it fails, it takes 35.5ms. On a peak day, Tor has about 1.85 million users [13]. To quantify the amount of computation required, we perform AES-CMAC and ECDSA signature generation in a tight loop and measure the throughput. It gives 27.8 operations per second per core. Thus, with nine directory servers, assuming each has eight cores, the total computation time for remote attestation of 1.85 million daily clients will be about 15.4 minutes. We believe this is a moderate resource requirement for the directory authority.

Overhead of key generation: Finally, we measure the overhead of in-enclave key generation for two RSA key pairs: identity key (3072-bit) and signing key (2048-bit) for the directory authority. SGX-Tor takes 12% longer than the vanilla Tor (2.90 vs. 2.59 ms), including the time for sealing keys and unsealing for key recovery.

6.3 Compatibility and Deployability

We demonstrate the compatibility of SGX-Tor by admitting a long-running SGX-Tor relay into the existing Tor network as a middle relay. For comparison, we also run a vanilla Tor side-by-side. We compare both relays in terms of (a) network I/O bandwidth per second, (b) probability to be selected as a middle relay, and (c) advertised bandwidth obtained from a published consensus document from CollectTor [4]. The bandwidth statistics are averaged over a 30-minute window. Figure 11 shows the result obtained for two weeks. In total, SGX-Tor served 10.5 GB of traffic. Both relays obtained fast and stable flag on the same day. The average advertised bandwidth of SGX-Tor relay is 119 KB/s. We see that SGX-Tor is compatible with the existing Tor network, and for all metrics SGX-Tor shows a similar tendency with the vanilla Tor.

7 Discussion

Deployment issues: The simplest way to deploy SGX-Tor in the existing Tor ecosystem incrementally is to use cloud platforms. Tor already provides default VM images to run bridges on Amazon EC2 cloud [13]. When SGX becomes available on cloud platforms, we envision a similar deployment scenario for SGX-Tor. As a recent patch provides support for SGX virtualization [9], we believe that deployment of SGX-Tor using cloud platform is feasible. Note that incremental deployment involves in security tradeoffs, as not all properties can be achieved as discussed in §4. As a future work, we would like to quantify the security tradeoffs and ways to mitigate attacks in the presence of partial deployment.

Limitation: Although SGX-Tor can mitigate many attacks against Tor components, attacks assuming network-level adversaries [44] and Sybil attacks [36] are still effective, as we mentioned in the §3.1. Additionally, SGX-Tor cannot validate the correctness of the enclave code itself. SGX-Tor can be compromised if the code contains software vulnerabilities and is subject to controlled side-channel attacks [69]. We believe these attacks can be

mitigated by combining work from recent studies: e.g., by checking whether an enclave code leaks secrets [65]; by protecting against side-channel attacks [62, 64]; or by leveraging software fault isolation [38, 62]).

8 Related Work

Software for trusted execution environments: Various TEEs such as TPM, ARM TrustZone, and AMD SVM have been used for guaranteeing the security of applications in mobile and PC environments. Since the traditional trusted computing technologies (e.g., hypervisor-based approach with TPM) rely on the chain of trust, it makes the size of TCB larger. Flicker [52] proposed an approach that executes only a small piece of code inside the trusted container, where it extremely reduces the TCB. Nevertheless, it suffers from performance limitations. Intel SGX removes this challenge by offering native performance and multi-threading. In addition, the cloud computing and hosting service providers, where Tor relays are often hosted [14], is predominantly x86-based.

Applications for Intel SGX: Haven [26] pioneered adopting Intel SGX in the cloud environment with an unmodified application. VC3 [61] proposed data analytics combined with Intel SGX in the cloud. Moat [65] studied the verification of application source code to determine whether the program actually does not leak private data on top of Intel SGX. Kim et al. [46] explores how to leverage SGX to enhance the security and privacy of network applications. These studies are early studies of SGX that rely on SGX emulators [42]. S-NFV [63] applies SGX to NFV to isolate its state from the NFV infrastructure and platform and presents preliminary performance evaluations on real SGX hardware. In contrast, we show how SGX can improve the trust model and operation of Tor and SGX-Tor run on real SGX hardware.

Attacks and security analysis on Tor: §3 discussed many attacks on Tor. SGX-Tor prevents modification of Tor binaries and limits attackers' ability; attackers can still launch attacks within the network outside Tor nodes. For example, attackers can still mount traffic analysis attacks or website finger-printing attacks. While Tor does not try to protect against traffic confirmation attacks [35], it aims to protect against general traffic analysis attacks. In particular, large-scale traffic correlation [29] and website finger-printing [45] attacks are believed to be very difficult in practice [22, 45] because those attacks require achieving an arbitrarily low false positive rate as the number of users becomes larger. Security analysis of Tor on a realistic workload is an ongoing research [34] area. In this work, we show how we can thwart known attacks against the Tor ecosystem by using a commodity trusted execution environment, Intel SGX.

9 Conclusion

Due to the wide adoption of the x86 architecture, Intel Software Guard Extensions (SGX) potentially has a tremendous impact on providing security and privacy for network applications. This paper explores new opportunities to enhance the security and privacy of a Tor anonymity network. Applying SGX to Tor has several benefits. First, we show that deploying SGX on Tor can defend against known attacks that manipulate Tor components. Second, it limits the information obtained by running or compromising Tor components, reducing the power of adversaries to network-level adversaries, who can only launch attacks external to the Tor components. Finally, this brings changes to the trust model of Tor, which potentially simplifies Tor operation and deployment. Our extensive evaluation of the SGX-Tor shows that SGX-enabled Tor components incur small performance degradation and supports incremental deployment on the existing Tor network, demonstrating its viability.

10 Acknowledgment

We thank the anonymous reviewers and our shepherd Phillipa Gill for their feedback. We also thank Will Scott, Chris Magistrado, Hyeontaek Lim, and Inho Choi for their comments on earlier versions. Dongsu Han is the corresponding author. This work was supported in part by the IITP funded by the Korea government (MSIP) [B0101-16-1368, Development of an NFV-inspired networked switch and an operating system for multi-middlebox services] and [B0190-16-2011, Korea-US Collaborative Research on SDN/NFV Security/Network Management and Testbed Build]; Office of Naval Research Global (ONRG); and NSF awards DGE-1500084, CNS-1563848, and CRI-1629851.

References

[1] Alexa: The top 500 sites on the web. `http://www.alexa.com/topsites`.

[2] Amazon Web Service. `https://aws.amazon.com/`.

[3] Atlas, flag:authority. `https://atlas.torproject.org/#search/flag:authority`.

[4] CollecTor. `https://collector.torproject.org/`.

[5] FBI's use of Tor exploit is like peering through "broken blinds". `http://arstechnica.com/tech-policy/2016/06/fbis-use-of-tor-exploit-is-like-peering-through-broken-blinds/`. Last accessed: Aug 2016.

[6] Intel Software Guard Extensions (Intel SGX) SDK. https://software.intel.com/en-us/sgx-sdk. Last accessed: Aug 2016.

[7] libevent-2.0.22. http://libevent.org/.

[8] OpenSSL-1.1.0. https://www.openssl.org/.

[9] Sgx virtualization. https://01.org/intel-software-guard-extensions/sgx-virtualization.

[10] Tor bandwidth scanner. https://trac.torproject.org/projects/tor/attachment/ticket/2861/bwauth-spec.txt.

[11] Tor: Bridges. https://www.torproject.org/docs/bridges.html.en. Last accessed: May, 2016.

[12] Tor Browser. https://www.torproject.org/projects/torbrowser.html.en.

[13] Tor Metrics. https://metrics.torproject.org/.

[14] Tor Project: GodBadISPs. https://trac.torproject.org/projects/tor/wiki/doc/GoodBadISPs.

[15] Tor:Overview. https://www.torproject.org/about/overview. Last accessed: Aug 2016.

[16] zlib-1.2.8. http://www.zlib.net/.

[17] Research problem: measuring the safety of the tor network. https://blog.torproject.org/blog/research-problem-measuring-safety-tor-network, 2011.

[18] Research problems: Ten ways to discover Tor bridges. https://blog.torproject.org/blog/research-problems-ten-ways-discover-tor-bridges, Oct 2011.

[19] Extend OOM handler to cover channels/connection buffers. https://trac.torproject.org/projects/tor/ticket/10169, 2014.

[20] How to report bad relays. https://blog.torproject.org/blog/how-report-bad-relays, July 2014.

[21] Tor security advisory: "relay early" traffic confirmation attack. https://blog.torproject.org/blog/tor-security-advisory-relay-early-traffic-confirmation-attack, July 2014.

[22] Traffic correlation using netflows. https://blog.torproject.org/category/tags/traffic-confirmation, Nov 2014.

[23] The chutney tool for testing and automating Tor network setup. https://gitweb.torproject.org/chutney.git, 2015. Accessed: 09/01/2016.

[24] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Proc. HASP*, pages 1–8, Tel-Aviv, Israel, 2013.

[25] K. Bauer, D. McCoy, D. Grunwald, T. Kohno, and D. Sicker. Low-resource Routing Attacks Against Tor. In *Proc. ACM Workshop on Privacy in Electronic Society*, 2007.

[26] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proc. USENIX OSDI*, 2014.

[27] A. Biryukov, I. Pustogarov, and R.-P. Weinmann. Trawling for Tor Hidden Services: Detection, Measurement, Deanonymization. In *Proc. IEEE Symposium on Security and Privacy*, pages 80–94, 2013.

[28] N. Borisov, G. Danezis, P. Mittal, and P. Tabriz. Denial of Service or Denial of Security? In *Proc. ACM CCS*, 2007.

[29] S. Chakravarty, M. V. Barbera, G. Portokalidis, M. Polychronakis, and A. D. Keromytis. On the effectiveness of traffic analysis against anonymity networks using flow records. In *Proc. Passive and Active Measurement*, 2014.

[30] S. Checkoway and H. Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proc. ASPLOS*, 2013.

[31] V. Costan and S. Devadas. Intel sgx explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint. iacr. org/2016/086.

[32] R. Dingledine. Tor Project infrastructure updates in response to security breach. http://archives.seul.org/or/talk/Jan-2010/msg00161.html, January 2010.

[33] R. Dingledine. Turning funding into more exit relays. https://blog.torproject.org/blog/turning-funding-more-exit-relays, July 2012.

[34] R. Dingledine. Improving Tor's anonymity by changing guard parameters. https://blog.torproject.org/blog/improving-tors-anonymity-changing-guard-parameters, Oct 2013.

[35] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-generation Onion Router. In *Proc. USENIX Security Symposium*, 2004.

[36] J. R. Douceur. The Sybil Attack. In *Proc. IPTPS*, 2002.

[37] N. S. Evans, R. Dingledine, and C. Grothoff. A Practical Congestion Attack on Tor Using Long Paths. In *Proc. USENIX Security Symposium*, Berkeley, CA, USA, 2009. USENIX Association.

[38] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan:A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proc. USENIX OSDI*, 2016.

[39] Intel. Intel Software Guard Extensions Programming Reference (rev1), Sept. 2013. 329298-001US.

[40] Intel. Intel Software Guard Extensions Programming Reference (rev2), Oct. 2014. 329298-002US.

[41] Intel. Intel Software Guard Extensions Remote Attestation End-to-End Example. `https://software.intel.com/en-us/articles/intel-software-guard-extensions-remote-attestation-end-to-end-example?language=de`, July 2016.

[42] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han. OpenSGX: An Open Platform for SGX Research. In *Proc. NDSS*, San Diego, CA, Feb. 2016.

[43] R. Jansen, F. Tschorsch, A. Johnson, and B. Scheuermann. The Sniper Attack: Anonymously Deanonymizing and Disabling the Tor Network. In *Proc. NDSS*, 2015.

[44] A. Johnson, C. Wacek, R. Jansen, M. Sherr, and P. Syverson. Users Get Routed: Traffic Correlation on Tor by Realistic Adversaries. In *Proc. ACM CCS*, pages 337–348. ACM, 2013.

[45] M. Juarez, S. Afroz, G. Acar, C. Diaz, and R. Greenstadt. A Critical Evaluation of Website Fingerprinting Attacks. In *Proc. CCS*, 2014.

[46] S. Kim, Y. Shin, J. Ha, T. Kim, and D. Han. A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications. In *Proc. ACM Workshop on Hot Topics in Networks*, page 7. ACM, 2015.

[47] A. Kwon, M. AlSabah, D. Lazar, M. Dacier, and S. Devadas. Circuit fingerprinting attacks: passive deanonymization of Tor hidden services. In *Proc. USENIX Security Symposium*, pages 287–302, 2015.

[48] S. Le Blond, P. Manils, A. Chaabane, M. A. Kaafar, C. e. Castelluccia, A. Legout, and W. Dabbous. One Bad Apple Spoils the Bunch: Exploiting P2P Applications to Trace and Profile Tor Users. In *Proc. USENIX Conference on Large-scale Exploits and Emergent Threats*, pages 2–2, 2011.

[49] J. Lenhard, K. Loesing, and G. Wirtz. Performance Measurements of Tor Hidden Services in Low-Bandwidth Access Networks. In *International Conference on Applied Cryptography and Network Security*, pages 324–341, 2009.

[50] Z. Ling, J. Luo, W. Yu, X. Fu, D. Xuan, and W. Jia. A New Cell-Counting-Based Attack Against Tor. *IEEE/ACM Transactions on Networking (TON)*, 20(4):1245–1261, 2012.

[51] K. Loesing. *Privacy-enhancing technologies for private services.* PhD thesis, University of Bamberg, 2009.

[52] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proc. EUROSYS*, pages 315–328, 2008.

[53] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proc. HASP*, pages 1–8, Tel-Aviv, Israel, 2013.

[54] P. Mittal, A. Khurshid, J. Juen, M. Caesar, and N. Borisov. Stealthy Traffic Analysis of Low-latency Anonymous Communication Using Throughput Fingerprinting. In *Proc. ACM CCS*, pages 215–226, New York, NY, USA, 2011. ACM.

[55] S. J. Murdoch and G. Danezis. Low-Cost Traffic Analysis of Tor. In *Proc. IEEE Symposium on Security and Privacy*, pages 183–195, Washington, DC, USA, 2005. IEEE Computer Society.

[56] S. J. Murdoch and P. Zieliński. Sampled Traffic Analysis by Internet-exchange-level Adversaries. In *Proc. Privacy Enhancing Technologies*, Berlin, Heidelberg, 2007. Springer-Verlag.

[57] L. Overlier and P. Syverson. Locating Hidden Servers. In *Proc. IEEE Symposium on Security and Privacy*, pages 100–114, 2006.

[58] V. Pappas, E. Athanasopoulos, S. Ioannidis, and E. P. Markatos. Compromising Anonymity Using Packet Spinning. In *Proc. Information Security Conference*, pages 161–174, 2008.

[59] M. Perry. A Critique of Website Traffic Fingerprinting Attacks. `https://blog.torproject.org/blog/critique-website-traffic-fingerprinting-attacks`, Oct 2013.

[60] R. Pries, W. Yu, X. Fu, and W. Zhao. A New Replay Attack Against Anonymous Communication Networks. In *Proc. IEEE International Conference on Communications*, pages 1578–1582, May 2008.

[61] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Proc. IEEE Security and Privacy (SP)*, pages 38–54. IEEE, 2015.

[62] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *Proc. NDSS*, 2017.

[63] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska. S-NFV: Securing NFV States by Using SGX. In *Proc. ACM International Workshop on Security in SDN-NFV*, 2016.

[64] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proc. NDSS*, 2017.

[65] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani. Moat: Verifying confidentiality of enclave programs. In *Proc. ACM CCS*, pages 1169–1184, 2015.

[66] Y. Sun, A. Edmundson, L. Vanbever, O. Li, J. Rexford, M. Chiang, and P. Mittal. RAPTOR: Routing Attacks on Privacy in Tor. In *Proc. USENIX Security Symposium*, 2015.

[67] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter. Cooperation and Security Isolation of Library OSes for Multi-Process Applications. In *Proc. European Conference on Computer Systems*, page 9. ACM, 2014.

[68] P. Winter, R. Köwer, M. Mulazzani, M. Huber, S. Schrittwieser, S. Lindskog, and E. Weippl. Spoiled onions: Exposing malicious tor exit relays. In *Proc. Privacy Enhancing Technologies*, pages 304–331. Springer, 2014.

[69] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proc. IEEE Symposium on Security and Privacy*, 2015.

Appendix A

Type	Category	API	Description
ECALL	TOR	`void sgx_start_tor(int argc, char **argv, ...)`	Start SGX-Tor process in enclave
ECALL	TOR	`void sgx_start_gencert(char *tor_cert, ...)`	Create authority keys and certificate
ECALL	TOR	`void sgx_start_fingerprint(char *fingerprint, ...)`	Create finger-print
ECALL	TOR	`void sgx_start_remote_attestation_server(int port, ...)`	Start remote attestation server
ECALL	TOR	`sgx_status_t sgx_init_ra(int b_pse, ...)`	Create remote attestation context
ECALL	TOR	`sgx_status_t sgx_close_ra(sgx_ra_context_t context)`	Release remote attestation context
ECALL	TOR	`sgx_status_t sgx_verify_att_result_mac(sgx_ra_context_t context, ...)`	Verify the MAC in attestation result
OCALL	NET	`int ocall_sgx_socket(int af, int type, int protocol)`	Create socket descriptor
OCALL	NET	`int ocall_sgx_bind(int s, const struct sockaddr *addr, ...)`	Bind socket
OCALL	NET	`int ocall_sgx_listen(int s, int backlog)`	Make socket in listening state
OCALL	NET	`int ocall_sgx_accept(int s, struct sockaddr *addr, ...)`	Accept incoming connection
OCALL	NET	`int ocall_sgx_send(int s, char *buf, int len, int flags)`	Receive data from given socket
OCALL	NET	`int ocall_sgx_recv(int s, char *buf, int len, int flags)`	Send data through given socket
OCALL	NET	`int ocall_sgx_select(int nfds, void *rfd, ..., struct timeval *timeout, int tv_size)`	Examining the status of socket descriptor
OCALL	THREAD	`unsigned long long ocall_sgx_beginthread(void *args, ...)`	Create thread with given argument
OCALL	THREAD	`unsigned long ocall_sgx_TlsAlloc(void)`	Allocates a thread local storage index
OCALL	ERROR	`int ocall_sgx_GetLastError(void)`	Get the error code of calling thread
OCALL	ERROR	`void ocall_sgx_SetLastError(int e)`	Set the error code of calling thread
OCALL	MEM	`int ocall_sgx_CreateFileMapping(int hFile, ...)`	Create file mapping object
OCALL	MEM	`void * ocall_sgx_MapViewOfFile(int hFileMappingObject, ...)`	Mapping address space for a file

Table 4: Interface between the Tor enclave and the privilege software. SGX-Tor uses seven ECALLS to bootstrap the Tor-enclave and aid remote attestation. It uses a total of 57 OCALLs in four categories. We list representative OCALLs in each category.

ViewMap: Sharing Private In-Vehicle Dashcam Videos

Minho Kim, Jaemin Lim, Hyunwoo Yu, Kiyeon Kim, Younghoon Kim, Suk-Bok Lee

Hanyang University

Abstract

Today, search for dashcam video evidences is conducted manually and its procedure does not guarantee privacy. In this paper, we motivate, design, and implement ViewMap, an automated public service system that enables sharing of private dashcam videos under anonymity. ViewMap takes a profile-based approach where each video is represented in a compact form called a view profile (VP), and the anonymized VPs are treated as entities for search, verification, and reward instead of their owners. ViewMap exploits the line-of-sight (LOS) properties of dedicated short-range communications (DSRC) such that each vehicle makes VP links with nearby ones that share the same sight while driving. ViewMap uses such LOS-based VP links to build a map of visibility around a given incident, and identifies VPs whose videos are worth reviewing. Original videos are never transmitted unless they are verified to be taken near the incident and anonymously solicited. ViewMap offers untraceable rewards for the provision of videos whose owners remain anonymous. We demonstrate the feasibility of ViewMap via field experiments on real roads using our DSRC testbeds and trace-driven simulations.

1 Introduction

A dashcam is an onboard camera that continuously records the view around a vehicle (Fig. 1). People install dashcams in their vehicles because irrefutable video evidence can be obtained in the event of an accident. Dashcams are becoming popular in many parts of Asia and Europe. For example, the rate of dashcam adoption has exceeded 60% in South Korea [1]. Other countries with high adoption rates include Russia and China. In these nations, the use of dashcams has now become an integral part of the driving experience of individuals.

Dashcams, as a side benefit, have tremendous potential to act as silent witnesses to others' accidents. Authorities such as police want to exploit the potential of dashcams because their videos, if collected, can greatly assist in the accumulation of evidence, providing a complete picture of what happened in incidents. For example, the police appeal for public to send in dashcam video evidences of certain traffic accidents [2, 3].

Figure 1: Dashcam installation on the windshield [4].

However, the current practice of volunteer-based collection limits participation of the general public due to the following reasons. First, people are reluctant to share their videos in fear of revealing their location history. Users want strong *anonymity*. Second, people are not interested in things that do not specifically concern them. Users want some form of *compensation* for provision of their videos. Third, people do not like to be annoyed by manual procedures, e.g., checking a wanted list, searching through their own videos, and sending in the matched videos all by themselves. Users want *automation* for hassle-free participation.

In this work, we aim to build a system that fulfills the key requirements above. There are, however, three challenges on the way. First, the authenticity of videos must be validated under users' anonymity. Verification of locations and times of videos should not rely on existing infrastructure such as 3G/4G networks where user identities may be exposed. Second, verifiable reward must be given without tracking users. The difficulty lies in making the reward double-spending proof while not linking it to users' videos. Third, irrelevant videos must be automatically rejected without human review. Under anonymity, attackers may simply upload an overwhelming number of fake videos, making it impractical to manually examine all by human eyes.

In this paper, we present ViewMap, a public service system (run by authorities) that addresses the challenges of dashcam sharing. To preserve user privacy, ViewMap takes a profile-based approach where each video is represented in a compact form called a *view profile* (VP), and the anonymized VPs are treated as entities for search, verification, and reward instead of their owners. Each VP makes verifiable links with nearby vehicles' VPs that share the same sight via line-of-sight (LOS) properties

of DSRC radios. This VP linkage process also incorporates inter-vehicle path obfuscation for protection against tracking. The system accumulates anonymized VPs (from normal users) and trusted VPs (from authorities, e.g., police cars), and uses such LOS-based VP links to build a map of visibility on a given incident in the form of a mesh-like structure called a *viewmap*. Its key strengths are that: (i) trusted VPs do not need to be near the incident location; (ii) the system can pinpoint VPs whose original videos are worth reviewing, and reject, if any, fake VPs cheating locations and/or times via the unique linkage structure; and (iii) it leads minimal communication overhead because original videos are never transmitted unless verified taken near the incident. Users upload videos only when anonymously solicited by the system for further human checking. The system rewards users with virtual cash that is made untraceable based on blind signatures.

We demonstrate the feasibility of ViewMap via field experiments on real roads with our DSRC testbeds and trace-driven simulations. Our evaluations show that ViewMap provides: (i) users with strong location privacy (tracking success ratio $< 0.1\%$); and (ii) the system with high-accuracy verification ($> 95\%$) in face of an extremely large number of fake VPs cheating locations and times.

Besides the privacy of users sharing videos, there exists an additional privacy concern specific to dashcam video sharing. Video contents may threaten the privacy of others visible in the videos. We do not completely handle this, but provide some defense for video privacy. Specifically, we have implemented the realtime license plate blurring, which is integrated into ViewMap-enabled dashcams. However, other sensitive objects can still be captured (e.g., pedestrians walking into the view). This video privacy is not fully addressed in this work, and merits separate research.

In summary, we make the following contributions:

1. **New application:** We introduce a new application that enables sharing of dashcam videos. It poses unique challenges: combination of location privacy, location authentication, anonymous rewarding, and video privacy at the same time.
2. **Comprehensive solution package:** We present a solution suite that finds, verifies, and rewards private, location-dependent dashcam video evidence by leveraging DSRC-based inter-vehicle communications without resorting to existing infrastructure where user identities may be exposed.
3. **Prototype and evaluation:** We build a full-fledged prototype and conduct real road experiments using ViewMap-enabled dashcams with DSRC radios. It validates that LOS-based VP links are strongly associated with the shared "view" of videos in reality. Our evaluations show that ViewMap achieves strong privacy protection and high verification accuracy.

2 Background

Dashboard camera. Dashcams, installed on the windshield of a vehicle, continuously record in segments for a unit-time (1-min default) and store them via on-board SD memory cards. Once the memory is full, the oldest segment will be deleted and recorded over. For example, with 64 GB cards, videos can be kept for 2-3 weeks with 1-2 hours daily driving. Dashcams feature a built-in GPS system that provides vehicle speed and location. Some dashcams have an advanced feature of a dedicated parking mode, where videos can be recorded when the motion detector is triggered, even if a vehicle is turned off.

Dashcam video sharing. The most prevalent way to obtain dashcam video evidences today is public announcement, and users voluntarily hand in their videos. Some organizations adopt a more arranged approach where dashcam users have themselves registered for a pool of volunteers. For example, the police in South Korea operate such a system called shadow cops [5], but the number of the registered users is only a very small proportion of dashcam users in the country, less than 0.01% as of 2016. Recent studies [6, 7] report that privacy concerns and monetary motives are two major factors behind the sharing of dashcam videos for urban surveillance.

3 Motivation

3.1 Use Cases

Analyzing traffic accidents. When investigating a traffic accident, dashcam videos recorded by nearby vehicles are often valuable. While an accident vehicle may have its own video, it only offers one partial view and does not guarantee conclusive evidence. Nearby vehicles, on the other hand, have wider views each with different angle on the accident. However, the major impediment, besides lack of systematic search, is that people are reluctant to share videos due to privacy concerns and the associated hassle without compensation.

Investigating crimes. Dashcams can assist crime investigations and also have great potential for crime prevention. While CCTV cameras are installed in public places [8], there exist a countless number of blind spots. Dashcams are ideal complements to CCTVs since pervasive deployment—cars are everywhere—is possible. However, the difficulty here is that users are not often aware whether they have such video evidences. This is because criminal evidences are not as noticeable as traffic accidents. Thus, the current practice of volunteer-based collection has more serious limitation in this context.

3.2 Threat Model

User privacy. Users face privacy risks when providing personal, location and time-sensitive information. The system may wish to track users via such collected location

samples. Time-series analysis on location samples, even if anonymized, could accumulate path information (i.e., following the footsteps) and eventually identify users location history [9, 10]. Such location tracking can further reveal users' very private information if the system can connect specific individuals to specific locations [11]. For example, an anonymous user at a hospital or other private location can be eventually identified if the user's path is tracked from his/her home (as a resident of a particular address). Besides, users face risk of revealing their identities and past locations when rewarded for their videos.

System security. On the other hand, if anonymity is provided, the system can becomes a target for various attacks. Dishonest users may claim rewards for fake videos cheating locations and/or times, or they may even fabricate video evidence. Human review may help identify and reject such videos. However, a manual review not only takes time, but also requires trained manpower. Anonymous attackers may launch denial-of-service attacks by simply uploading an overwhelming number of fake or dummy videos. Given limited resources of manpower, a manual review of all such videos would be impractical.

4 ViewMap Framework

We first highlight the key features of ViewMap (Fig. 2).

Visual anonymization. Vehicles perform license plate blurring on their video stream while recording. Only such content anonymized videos are used in ViewMap, and hereafter referred to simply as "videos".

Profile-based anonymity. Each video (1-min default) is represented by its *view profile*, VP, that summarizes (i) time/location trajectory, (ii) video fingerprint, and (iii) fingerprints of videos taken by neighbor vehicles that share the same sight (via DSRC radios). We call such association between two neighbor VPs a *viewlink* (see Fig. 2). Vehicles anonymously upload their past VPs to the system whenever possible (e.g., WiFi or in-vehicle Internet access). These anonymized, self-contained VPs are stored in the VP database and collectively used for search and reward instead of their owners.

Location tracking protection. Vehicles also upload *guard VPs*, which are indistinguishable from actual VPs, to the system for protection against tracking. These guard VPs are created not for actual videos, but for plausible trajectories among neighbor vehicles such that their actual paths become indeterminable from the system's viewpoint. This is to guarantee location privacy in the VP database. Guard VPs are used only for path obfuscation, and vehicles delete them in their storage after submission.

Automated collection of video evidences. When video evidence is required, the system retrieves relevant anonymous VPs (from normal users) and trusted VPs (from authorities, e.g., police cars; not necessarily near

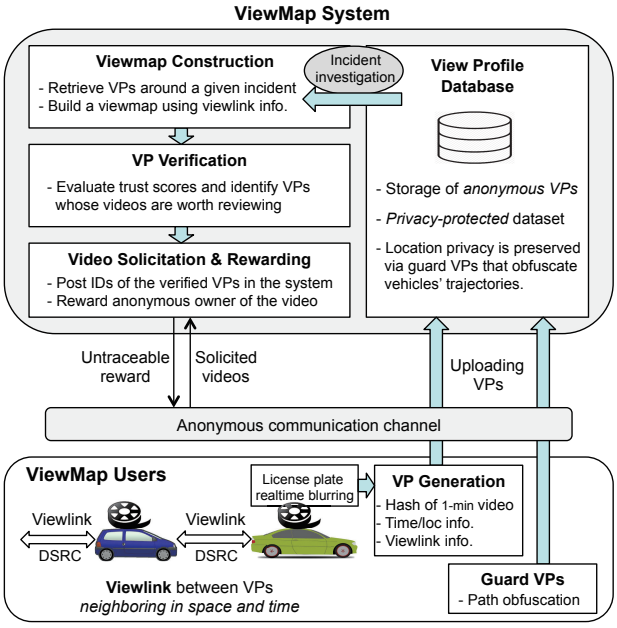


Figure 2: ViewMap framework.

the site), and builds viewmap(s) of a given incident using viewlinks among those VPs. The system exploits viewmap's unique structure to exclude, if any, fake VPs.[1] The verification criteria is: (i) 'trust scores' of the member VPs evaluated using trusted VPs as 'trust seeds' and (ii) their topological positions in the viewmap. The system solicits videos of the legitimate VPs (verified taken near the incident) by posting their VP identifiers,[2] without publicizing location/time of the investigation. Users upload anonymously the matched videos, if any. The videos are first validated against the system-owned VPs, and then reviewed by human investigators.

Untraceable reward. To reward users for provision of videos, their VP identifiers are likewise posted in the system. The users anonymously prove their ownership, and earn the virtual cash that is made untraceable via blind signatures. The authenticity of the virtual cash is self-verifiable, but never linked to users' VPs or videos.

5 Design of ViewMap

5.1 Protecting Location Privacy

5.1.1 Decoupling Users and Videos

Video recording. ViewMap-enabled dashcams are time-synched via GPS and continuously start recording new videos every minute on the minute. This is to facilitate the construction of viewmaps, each of which corre-

[1] We consider VPs (regardless of actual or guard VPs) that were created by users at their physical positions via the proper VP generation (Section 5.1) as *legitimate*; otherwise as *fake*.

[2] Both actual and guard VPs legitimately created near the incident may be on the requested video list, but the actual VPs only trigger video uploading. The guard VPs have already been deleted in users' storage.

sponds to a single unit-time (1-min) during an incident period. The recording procedure also performs license plate blurring in real time (detailed in Section 6.2) for video privacy (Fig. 3). The realtime plate blurring is chosen for two reasons: (i) post processing of videos, if allowed, opens the door for posterior fabrication of video evidence; and (ii) realtime visual anonymization can also alleviate visual privacy concerns[3] for the use of dashcams.

Video fingerprinting. Each vehicle, when completing 1-min recording of current video u, generates its unique view profile, VP_u that contains (i) time/location trajectory, (ii) fingerprints of u, and (iii) a summary (by a Bloom filter) of fingerprints of others' videos neighboring in space and time (via DSRC radios). To exchange fingerprints in the transient vehicular environment, each vehicle periodically (e.g., every second) produces and broadcasts a hash and associated information of its currently recording (for i secs: $1 \leq i \leq 60$) video u; we refer to such a cumulative fingerprint of u as *view digest*, VD_{u_i} (see Fig. 4). These VDs are exchanged between neighbors as below.

Broadcasting VDs. Every second, each vehicle A produces and broadcasts a view digest, VD_{u_i} of its video u currently being recorded for i secs using DSRC radio:

$$A \longrightarrow *: \quad T_{u_i}, L_{u_i}, F_{u_i}, L_{u_1}, R_u, H(T_{u_i}|L_{u_i}|F_{u_i}|H_{u_{i-1}}|u_i^{i-1}).$$

where T_{u_i}, L_{u_i}, and F_{u_i} are time, location, and byte-size of video u at i^{th} second, respectively. L_{u_1} is the initial location of u used for guard VP generation (Section 5.1.2). R_u is VP identifier of u. $H_{u_{i-1}}$ is the hash of previous $\text{VD}_{u_{i-1}}$, and u_i^{i-1} is a newly recorded content from $(i-1)^{\text{th}}$ to i^{th} seconds (see Fig. 4. Note: $H_{u_0} = R_u$). This cascaded hash operation facilitates the constant-time VD generation regardless of total file size. Note that original video u (within hash H) is not revealed in VD_u, and is later provided to the system only after VP_u is verified and anonymously solicited. The value R_u is further derived as $R_u = H(Q_u)$, where Q_u is file u's secret number chosen by A and is later used for untraceable rewarding.

Accepting neighbor VDs. Each vehicle A also receives VD_{x_j} broadcasted from nearby vehicle B:

$$B \longrightarrow *: \quad T_{x_j}, L_{x_j}, F_{x_j}, L_{x_1}, R_v, H(T_{x_j}|L_{x_j}|F_{x_j}|H_{x_{j-1}}|v_j^{j-1}).$$

where T_{x_j}, L_{x_j}, and F_{x_j} are time, location, and byte-size of video x at j^{th} sec, respectively. A first validates VD_{x_j} by checking whether T_{x_j} and L_{x_j} are in acceptable ranges: T_{x_j} within the current 1-sec interval; L_{x_j} inside a radius of DSRC radios. If yes, A treats VD_{x_j} as a valid one. A temporarily stores at most two valid VDs per neighbor: the first and the last received VDs with same R value.

[3]Recording using a dashcam is strongly discouraged in some countries such as Austria and Switzerland due to visual privacy concerns.

(a) Dashcam video 1. **(b)** Dashcam video 2.

Figure 3: Videos recorded by our ViewMap-enabled dashcams that perform license plate blurring in real time.

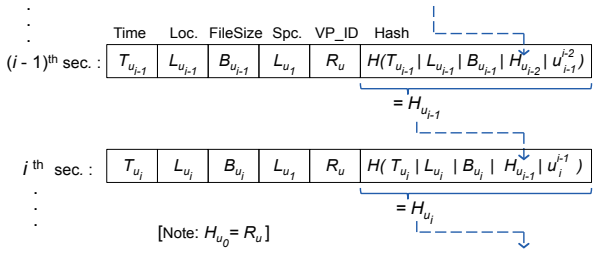


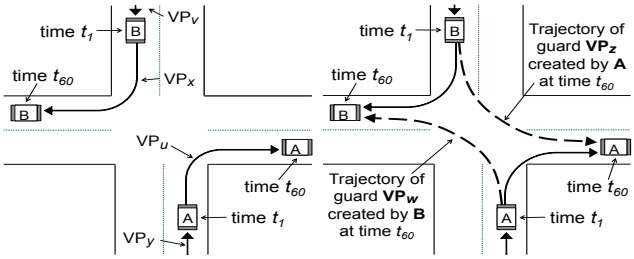
Figure 4: Example VDs of currently recording video u. A series of (sixty) VDs constitute a single (1-min) VP_u.

Generating actual VPs. Upon complete recording of current 1-min video u, each vehicle A generates its view profile VP_u. A first inserts the neighbor VDs (at most two VDs—the first and the last received VDs—per neighbor) into Bloom filter bit-array N_u. This reflects the neighbor VPs' cumulative fingerprints in part and their contact interval. A compiles its VDs (all VD_{u_i}: $1 \leq i \leq 60$) and Bloom filter bit-array N_u into VP_u, which will be anonymously uploaded to the system as described below.

5.1.2 Protection against Location Tracking

A collection of VPs, albeit each as anonymized location data, are subject to tracking. The system may follow a user's path by linking a series of VPs adjacent in space and time. For example in Fig.5a, the system tracking anonymous vehicle A can connect VP_y and VP_u using time-series location information. To obfuscate tracking, vehicles also generate and upload guard VPs as below.

Creating guard VPs. Each vehicle A generates guard VP(s) along with actual VP_u at the end of 1-min recording (e.g., time=t_{60} in Fig.5a). More specifically, among its m neighbor VPs, A randomly picks $\lceil \alpha \times m \rceil$ neighbor VPs ($0 < \alpha \leq 1$) and creates guard VPs for them (We use $\alpha = 0.1$, which is discussed in Section 6). For each chosen neighbor VP_x, A creates a guard VP (VP_z in Fig. 5b) whose trajectory starting at VP_x's initial location L_{x_1} (logged in its VDs) and ending at its own VP_u's last position. There are readily available on/offline tools that instantly return a driving route between two points on a road map. In this work, we make vehicles to use



(a) Actual VPs' trajectories. **(b)** Guard VPs's trajectories.

Figure 5: Protection against location tracking.

Google Directions API [12] for this purpose. In an effort to make guard VPs indistinguishable from actual VPs, we arrange their VDs variably spaced (within the predefined margin) along the given routes. Guard VPs are not for actual videos and thus, their hash fields are filled with random values. *A* makes neighborship between guard and actual VPs by inserting their VDs into each other's Bloom filter bit-arrays. *A* now clears all temporary memories, and resumes a new round of VP generation for the next recording video.

Cooperative privacy. Creating guard VPs for one another realizes continuously divergent paths from the viewpoint of observers, obfuscating the system's tracking of a vehicle's trajectory. Unlike the previous schemes like Mix-zones [13], this approach does not strictly require space-time intersections of vehicles' paths. Instead, vehicles can create path confusion any time, any place within DSRC range (up to 400 m). Note that they only need to be within DSRC range for a time to be considered as neighbors, not necessarily for the whole 1-min period.

Uploading VPs. Vehicles, whenever connected to the network (e.g., WiFi or in-vehicle Internet access), upload their actual and guard VPs anonymously to the system. We use Tor [14] for this purpose. More specifically, we make users constantly change sessions with the system, preventing the system from distinguishing among users by session ids. The submitted VPs are stored in the VP database of the system. Vehicles keep their actual VPs but delete guard VPs in their storage after submission.

5.2 Collecting Video Evidences

5.2.1 Viewmap Construction

In search of video evidences of an incident, the system builds a series of viewmaps each corresponding to a single unit-time (e.g., 1 min) during the incident period. We here describe the construction of a single viewmap at certain 1-min time interval at t, which is built only with VPs (actual and guard VPs)[4] whose times are t.

[4]From the system's perspective, actual and guard VPs are indistinguishable, thus treated equally. We hereafter refer to both as "VP".

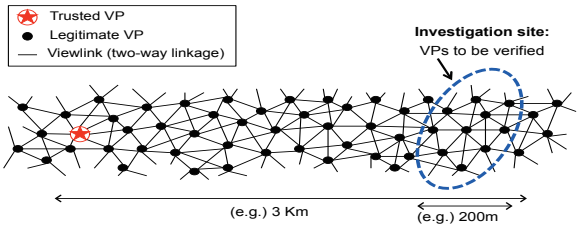


Figure 6: Illustration of a viewmap.

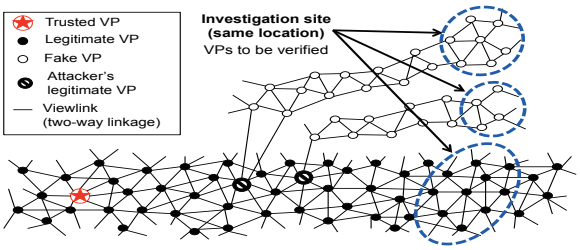


Figure 7: An example viewmap with fake VPs.

We first select the *trusted* VP_s closest to the investigation site l, where VP_s is a special VP from authorities such as police cars. The geographical coverage[5] of the viewmap spans an area C that encompasses l and VP_s (Fig. 6). Then, all the VPs whose claimed locations are inside area C (anytime during incident time t) become members of the viewmap. For each member VP_u, we find its neighbor candidates VP_v such that any of their time-aligned locations L_{u_i} and L_{x_j} are within the radius of DSRC radios. We then validate the mutual neighborship between VP_u and VP_v via membership queries (Bloom filter) using their element VDs on N_v and N_u. If none of the element VDs (of either VPs) passes the Bloom filter test, they are not mutual neighbor VPs. In this way, we build a viewmap by creating edges between 'two-way' neighbor VPs. We refer to such an edge as a *viewlink*, which signifies that two VPs are line-of-sight neighbors that share the same sight at some point in their times.

5.2.2 ViewProfile Verification

Given a constructed viewmap, there are certain VPs whose claimed locations (anytime during incident period) fall within an investigation site. The goal is to identify legitimate VPs among them. We consider VPs that join in a viewmap via the proper VP generation (as in Section 5.1) as *legitimate*; otherwise as *fake*.

Insights. A viewmap appears single-layered (Fig. 6) when all members are legitimate VPs. On the other hand, a viewmap with fake VPs results in multi-layered structure (Fig. 7), with only one layer containing a trust VP. This is because the validation of two-way linkage prevents attackers from creating arbitrary edges with other users' legitimate VPs without actual VD exchange. Thus,

[5]The coverage area of a viewmap is normally much larger than the size of the investigation site since trusted VPs (i.e., police cars) *were* not always very close to the site.

Algorithm 1 Verify_VPs(viewmap G)

```
Let M be the transition matrix on G
Let d be the trust distribution vector by the trusted VP
Let P be the trustRank-score vector, initially P = d
Let δ be a damping factor
// compute TrustRank scores
while not converged do
    P = δ·M·P + (1−δ)·d
end while
// determine the legitimacy of VPs
Let X be the all VPs within investigation site of G
Mark the highest scored VP u ∈ X as 'LEGITIMATE'
Let W(⊂ X) be VPs reachable from u only via X
for each VP i ∈ W do
    Mark i as 'LEGITIMATE'
end for
```

attackers can only inject fake VPs by linking them with their own legitimate VPs, if any. This is a very restrictive condition for attackers unless they happened to be actually in the vicinity of the incident. Moreover, the location proximity checking between neighbor VPs precludes long-distance edges and thus, forces attackers to create their own chain of fake VPs in order to place some of them in the target site.

Evaluating trust scores. To exploit the linkage structure of viewmaps, we adopt the TrustRank[6] algorithm tailored for our viewmap. In our case, a trusted VP (as a trust seed) has an initial probability (called trust scores) of 1, and distributes its score to neighbor VPs divided equally among all adjacent 'undirected' edges (unlike outbound links in the web model). Iterations of this process propagate the trust scores over all the VPs in a viewmap via its linkage structure. As shown in Algorithm 1, for a given viewmap G, trust scores P of all VPs are calculated via iterations of the following matrix operation:

$$\mathbf{P} = \delta \cdot \mathbf{M} \cdot \mathbf{P} + (1-\delta) \cdot \mathbf{d},$$

where $\mathbf{M}$ is the transition matrix representing VP linkage of G, $\mathbf{d}$ is the trust score distribution vector with an entry for trust VP only to 1, and δ is a damping factor empirically set to 0.8. The trust scores of all member VPs are obtained upon convergence of P.

Given a trusted VP z and an investigation site X, the VPs in X of z's layer are strongly likely to have higher trust scores than VPs in X of other layers. This is because the flow of trust scores has more chance of using edges within the base layer of z than using cross-layer edges (analyzed in Section 6). Accordingly, we identify the highest trust scored VP u in X as legitimate. All the VPs in X reachable from u strictly via VPs in X are determined legitimate as well.

[6]The TrustRank algorithm [15] outputs a probability distribution of likelihood that a person randomly clicking on links, starting from a certain 'seed' page, will arrive at any particular page.

5.2.3 Solicitation of Videos

Once VPs near a given incident is identified, the system solicits the videos. Owners are unknown and thus, they are requested via VP identifiers. More specifically, legitimate VP_u's identifier R_u is marked as 'request for video' and posted in the system. Users check the list of solicited videos when accessing the system, and upload anonymously, if any, the matched video u along with its VP_u. The video is first validated via cascading hash operations against the system-owned VP, and then reviewed by human investigators.

5.3 Rewarding Anonymous Users

After human review, the system offers untraceable virtual cash to reward users for provision of videos based on their contributions. The system S posts VP identifier R_u (of reviewed video u) marked as 'request for reward', and corresponding user A obtains from S the virtual cash that is made untraceable using blind signatures [16] as follows: (i) A provides secret number Q_u of video u ($R_u = H(Q_u)$) to prove its ownership, (ii) A makes a blinded message $B(H(m_u), r_u)$ using a blinding secret r_u, (iii) S signs the message with its private key K_S^- not knowing the 'blinded' content m_u, and (iv) A unblinds the signed message using blinding secret r_u. This unblinded signature-message pair $(\{H(m_u)\}_{K_S^-}, m_u)$ results in virtual cash. When A presents it for payment, anyone can verify (i) its authenticity via S' signature and (ii) its freshness via double-spending checking on m_u, but fails to link A with his/her video u. Even the system cannot derive the link between A's virtual cash and video u's blinded message without the blinding secret r_u (only known to A). A more detailed description of the above procedure is provided in the Appendix.

6 Analysis of ViewMap

6.1 Overhead of ViewMap

Communication. The VP generation involves inter-vehicle communication to exchange up-to-the-second VDs via DSRC broadcast. The format of a VD message includes time and location information (8 bytes each), file size (8 bytes), VP identifier (16 bytes), and a cascaded hash value (16 bytes). Excluding PHY- and MAC-layer headers, the length of our VD message is thus only 72 bytes—even can be piggybacked into a DSRC beacon whose size is as large as nearly 300 bytes [17]. Vehicles also communicate with the system, where original videos are never transmitted unless specifically solicited. Instead, VPs are only transmitted, whose sizes are negligibly small compared with those of videos as shown below.

Storage. Given a 1-min dashcam video, its VP consists of 60 VDs and one Bloom filter bit-array (256 bytes in our context). Each video also has its secret number of 8 bytes for untraceable rewarding. Thus, the total storage

Platform	Blur time	I/O time	Frame rate
Rasp. Pi 3 (1.2 GHz)	50.19 ms	49.32 ms	10 fps
iMac 2008 (2.4 GHz)	10.72 ms	41.78 ms	18 fps
iMac 2014 (4.0 GHz)	10.18 ms	20.44 ms	30 fps

Table 1: Frame rates of realtime license plate blurring.

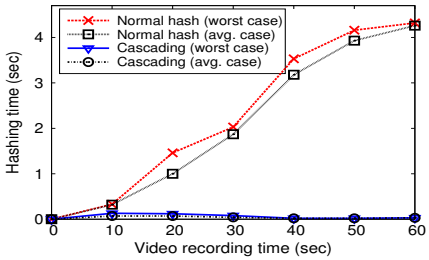


Figure 8: Hash generation times.

overhead incurred by a single VP is (60 × 72 bytes + 256 bytes + 8 bytes) = 4584 bytes. Given that the average size of 1-min video is 50 Mbytes, the storage overhead is less than 0.01% of original videos.

Computation. Two key operations that require computation are: car plate blurring (discussed shortly) and VD generation. To broadcast a new VD on time, a hash of the currently recording video should be produced within one second. Our test (Fig. 8), with Raspberry Pi (1.2 GHz CPU) as a dashcam for a 50-Mbyte 1-min video, shows that normal hashing times increase with recording time, and miss the deadline before 20 second of the video, reaching to 4.32 second at the end. On the other hand, our cascaded hashing results in constant-time hash generation with the worst-case of 0.13 second.

6.2 The Privacy of ViewMap

6.2.1 Visual Anonymity

Realtime license plate blurring. We have implemented the realtime license plate blurring using OpenCV [18] on Raspberry Pi as a ViewMap-enabled dashcam. The key task of license plate blurring is to localize plates in an image, a procedural part of popular car plate recognition algorithms [19, 20], whose realtime versions have been implemented for various mobile platforms such as iOS [21] and Android [22]. The license plate blurring procedure is as follows: (i) take the realtime video frame from camera module (**I/O time**), (ii) localize regions that contain license plates in the image via various parameters (e.g., area, aspect ratio)[7] and blur those areas (**Blur time**), (iii) write the plate blurred frame to the video file (**I/O time**). Our implementation on Raspberry Pi (1.2 GHz CPU) results in realtime processing with a frame rate of 10 fps (frame per second). Table 1 shows the time taken in each step when running our plate blurring implementation on Raspberry Pi as well as other platforms. We point out that the current prototype leaves more room for

[7]We use parameters tailored for South Korean license plates.

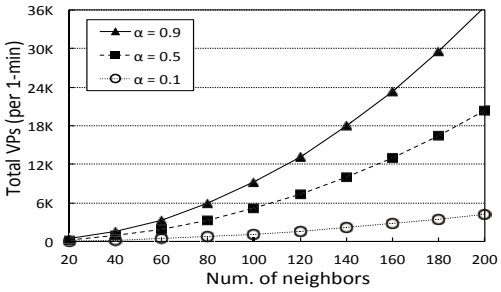


Figure 9: Volume of VP creation.

improvement, such as image processing using GPUs and multi-threading for blur and I/O operations. Still, our resulting videos at 10 fps, albeit frame rates not as high as normal movies (24-30 fps), are practically usable for the purpose of dashcams or video surveillance systems.

6.2.2 Location Privacy

The system can become a tracker using collected location traces in the VP database. We analyze the degree of privacy protection in the VP dataset against tracking.

Guard VPs against tracking. In order to create path confusion, each vehicle, among its m neighbor VPs, randomly selects $\lceil \alpha \times m \rceil$ neighbor VPs ($0 < \alpha \leq 1$) and generates guard VPs for them. Having a larger value of α creates higher degree of path confusion, but incurs excessively high volume of VPs in a dense environment (Fig. 9). Our design choice is to keep α small, but enough to preserve a high degree of privacy. The intuition behind is that driving time usually spans at least several minutes (10-min driving reported in [23]) so we exploit time factor t to have $P_t = \left[1 - \{1 - (1-\alpha)^m\}^m\right]^t$ below 0.01, where P_t is the probability that there is any vehicle not covered by others' guard VP until time t. In this work, we use $\alpha = 0.1$, which makes P_t below 0.01 with 5-min driving.

Tracking process. The tracker's process can be formalized through target tracking algorithms [24, 25]. In our VP-based traces, the tracker's prediction only happens at the end of the currently tracking VP to link to the next VP. We assume a strong adversary with perfect knowledge of the initial position of target vehicle u such that $p(u,0) = 1$, where $p(i,t)$ denotes the attacker's belief (probability) that location sample $l(i,t)$ of time t belongs to the vehicle currently tracked. At each VP start-time t, predicted position $l_u(t)$ of target u is given based on the last sample $l(j,t-1)$ of the previous VP. We derive the tracker's belief $p(i,t)$ of target u in $l(i,t)$ within possible distance from $l_u(t)$ based on [23], which follows a probability model of distance deviation from the prediction and this model ensures $\sum_i p(i,t) = 1$ for any time t.

Location entropy. To measure the degree of privacy under tracking, we use location entropy, defined as $H_t = \sum_i p(i,t) \log p(i,t)$, a quantitative measure of the tracker's

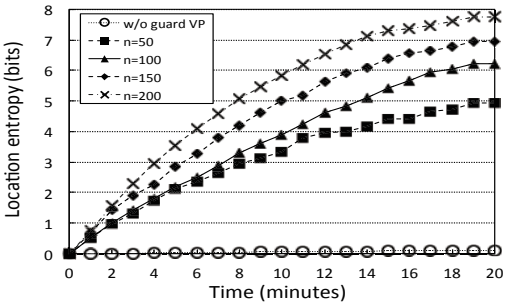


Figure 10: Location entropy.

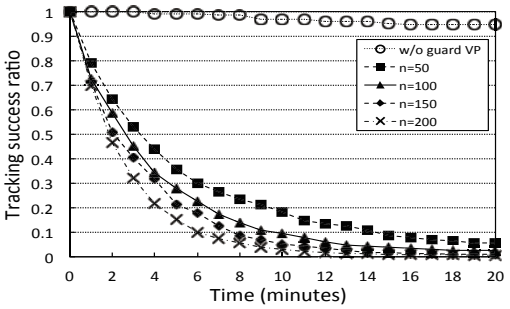


Figure 11: Tracking success ratio.

uncertainty [11, 23]. Here, H_t measures the uncertainty of correctly tracking a vehicle over time t. Lower values of H_t indicate more certainty or lower privacy.

Tracking success ratio. To give more intuitive privacy results, we also assess a tracking success ratio S_t, that measures the chance that the tracker's belief, when tracking target u over time t, is indeed true. Thus, S_t is equivalent to $p(u,t)$ of actual target u, since $\sum_i p(i,t) = 1$ at any time t. Note S_t is unknown to the tracker, who becomes unsure which $l(i,t)$ belongs to target u over time.

Privacy experiments. We collect a dataset of VPs created from 50-200 vehicles[8] traveling in the area of 4×4km^2 using *ns*-3 simulator. Fig 10 shows the average entropy over time. We also plot the entropy without guard VPs in the lowest density case ($n = 50$) as reference. Before ten minutes of driving, vehicles reach three bits of location entropy in the sparse case of $n = 50$. In other words, the tracker of a certain vehicle may suspect 8 different locations,[9] but without knowing the exact location. Fig 11 plots the average tracking success ratio over time. We see that, in the sparse case of $n = 50$, the tracking success ratio decreases to 0.2 before ten minutes and further drops below 0.1 before fifteen minutes. In the case without guard VPs, on the other hand, the tracking success ratio still remains above 0.9 even after twenty minutes. This result shows: (i) the privacy risk from anonymous location data in its raw form; and (ii) the privacy protection via guard VPs in the VP database.

[8] We present large-scale evaluations later in Section 8.

[9] X bits of entropy corresponds roughly to 2^X equally likely locations.

6.3 The Security of ViewMap

6.3.1 Attacks with a Large Number of Fake VPs

Attackers may submit fake VPs simply cheating locations and times. Such VPs are immediately excluded from a viewmap G unless linked to any legitimate 'member' VPs of G. Attackers must have physically positioned themselves in space and time on G to obtain legitimate VPs of G. This is a highly restrictive condition for attackers as they cannot predict the future investigation on G.

Let us nonetheless consider the case of attackers with such legitimate VPs on G. Under anonymity, the attackers can easily generate and inject a large number of fake VPs now all disguised as members of G. We further assume that the attackers share their fake VPs to increase their trust scores. To examine how ViewMap performs on such attacks, we provide our analysis on trust scores and our experiment results below.

Upper bounds of trust scores. Let $D_L(v)$ be the the set of VPs that are reachable from VP v by following at least L links. Similarly, we refer to the set of VPs that are distant from all VPs in a given group $\mathcal{G}$ by at least L links as $D_L(\mathcal{G})$ (i.e., $D_L(\mathcal{G}) = \cap_{v\in\mathcal{G}} D_L(v)$). By adopting Theorems 1 and 2 in [26], we obtain the upper bound for the sum of trust scores over $D_L(\mathcal{G})$ as follows.

Lemma 1. *Given a set T of trusted VPs, the sum of the trust scores over all VPs in $D_L(T)$ is at most α^L. That is, $\sum_{v\in D_L(T)} P_v \le \alpha^L$ where P_v is the trust score of v.*

Due to the above lemma, a VP not closer to a given trusted VP than L links cannot have a trust score greater than α^L. It implies that a trust score decreases by at most the ratio α to the minimum distance to a given trusted VP.

We next provide the upper bound of the sum of trusted scores over all fake VPs in the following lemma.

Lemma 2. *Let A and F_A denote a set of attackers and the set of fake VPs, generated by attackers in A to try a colluding attack, respectively. The sum of trust scores over F_A is upper bounded as*

$$\sum_{v\in F_A} P_v \le \frac{\alpha}{1-\alpha} \sum_{v\in A} \frac{|O_v \cap F_A|}{|O_v|} P_v \qquad (1)$$

where O_v represents the neighbors of v.

The proof is given in the Appendix.

The above lemma suggests that attackers may achieve high trust scores for fake VPs if attackers obtain high trust scores for their legitimate VPs. Considering Lemma 1, attackers can increase the chance of success in attacks if they position their legitimate VPs close to a given trusted VP. Furthermore, it shows that attackers will connect as many links to fake VPs as possible to increase the term $\frac{|O_v\cap F_A|}{|O_v|}$ in Inequation (1). Thus, the best strategy for an

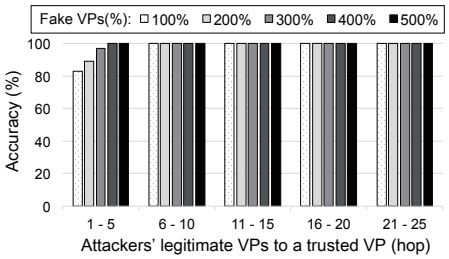


Figure 12: Verification result 1.

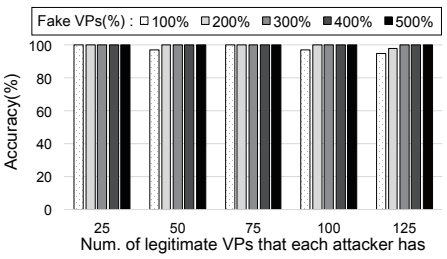


Figure 13: Verification result 2.

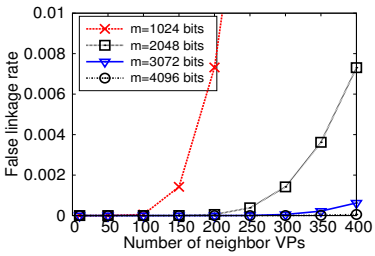


Figure 14: False linkage rate.

attacker is to connect every fake VP with its legitimate VP directly (while not practically possible in our viewmap), and the intuition leads us to the following corollary.

Corollary 1. *Assuming that attackers in collusion generate and inject n fake VPs, the upper bound of trust score of a fake VP positioned at the investigate site of a viewmap is* $\frac{1}{n}\frac{\alpha}{1-\alpha}\sum_{v\in A}\frac{|O_v\cap F_A|}{|O_v|}P_v$.

The above corollary suggests that injecting a large number of fake VPs is somewhat counterintuitively disadvantageous to attackers since their trust scores are distributed over all fake VPs—the more fake VPs, the lower their trust scores. Nevertheless, attackers have to create many fake VPs to spread them over a wide area so that some can reach the investigation site (publicly unknown), which in turn decreases the chances of their success in attacks.

Verification experiments. We run experiments on synthetic geometric graphs, as viewmaps with 1000 legitimate VPs. We use $5-15\%$ "human" colluding attackers injecting fake VPs that outnumber legitimate VPs up by 500% (i.e., 5000 fake VPs). Fig. 12 shows the results where accuracy indicates the cases that we correctly identify legitimate VPs during 1000 runs of each test. We see that, only the attackers whose legitimate VPs are very close to the trusted VP have the chance, albeit slight, of successful attacks (Lemma 2) while injecting many fake VPs rather decreases their chances (Corollary 1). In the experiments, accuracy is nearly 99% except the case of attackers in vicinity of the trusted VP (83% at worst). We however argue that such a case is rare since attackers cannot predict the future, e.g., location/time of an incident. We also consider another type of attacks where attackers prepare a lot of dummy videos beforehand and use them to obtain many legitimate VPs for a single viewmap. Fig. 13 shows that, under such a condition, accuracy is still high above 95%. This is because the trust scores of attackers are upperbounded by their topological positions within viewmap's linkage structure (out of their control) rather than their quantity.

6.3.2 False Linkage Attacks

ViewMap uses Bloom filters to validate VP linkage due to its compact size. One key problem with Bloom filters is false positives, leading to false linkage. ViewMap requires a low false linkage rate to prevent incorrect links. Given our two-way neighborship checking, the probability of false linkage is calculated as follows: $p=\left(1-\left[1-\frac{1}{m}\right]^{2nk}\right)^{2k}$ where m is a bit-array size, n and k are the number of neighbor VPs and the number of hash functions, respectively. Fig. 14 shows the false linkage rate using an optimal number of hash functions $k=(m/n)\ln 2$. Considering the maximum possible number of vehicles encountered in 1 minute, we choose to use m=2048 bits for our implementation. This has a false linkage rate of 0.1% with 300 neighbor VPs.

Attackers may fabricate their VPs' (Bloom filter) bit-arrays with all 1, claiming that they are neighbors of all other VPs. However, the validation of locations/times between VPs as well as the two-way neighbor checking prevent such attacks. Attackers may also try to poison neighbor VPs' (Bloom filter) bit-arrays to all 1, by sending out extremely many dummy VDs each associated with different VP while driving. Such a poisoning attack can be mitigated by limiting the number of neighbor VPs at each vehicle.[10]

7 Experiments on Real Roads

7.1 Measurement Framework

Our field measurement aims to provide answers to the following questions:

- Does our VP linkage reflect a line-of-sight (LOS) characteristic in reality?
- What are the implications of such LOS properties on (two-way) linked VPs and their videos?

Testbed implementation. Our DSRC testbed consists of on-board units (deployed in a rear window), Raspberry Pis with camera module (as dashcams), and Galaxy S5 phone (as a bridging device) as depicted in Fig. 18. We make Raspberry Pis as ViewMap-enabled dashcams to generate VDs of currently recording videos, which connects (via a Ethernet cable) to the DSRC OBU for VD broadcast, also connects (via Bluetooth) to the Galaxy phone installed with a Tor client [27] to anonymously upload its past VPs to our server.

[10] We set the maximum number of neighbor VPs accepted at each vehicle as 250 (neighbor vehicles) at this moment.

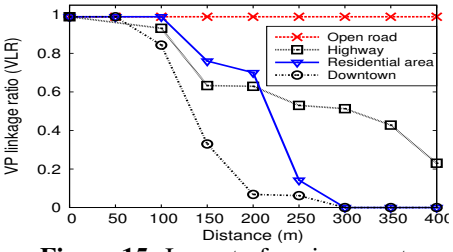


Figure 15: Impact of environment.

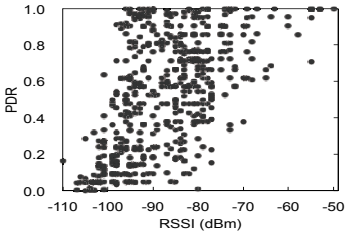


Figure 16: RSSI vs. PDR.

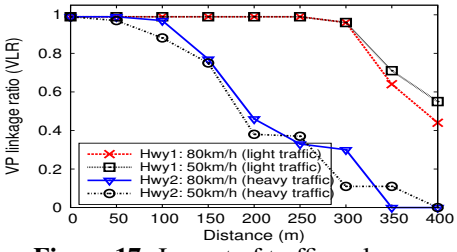


Figure 17: Impact of traffic volume.

Figure 18: Pictures of our in-vehicle setup.

Experiment settings. We conduct our experiments for three weeks in the metropolitan area of Seoul, Korea. We run experiments using two vehicles equipped with our ViewMap-enabled DSRC testbeds under various environments: university campus, downtown, residential area, and highways. Each vehicle, continuously recording the real view using a Raspberry Pi's camera (with realtime processing of car plate blurring), sends out VD broadcast messages every second. The transmission power is set to 14 dBm as recommended in [17]. In our experiments, we make vehicles only upload to our server their actual VPs in order to facilitate analyzing the relationship between linked VPs and their videos.

7.2 Measurement Results

7.2.1 Clear Manifestation of Line-of-Sight Links

Fig. 15 shows the measurement result of VP linkage ratio (VLR) on various environments. In the figure, VLRs of the open road environment are consistently very high (> 99%), indicating that *the range of VP linkage extends up to 400m if no obstacle is on the way*. Whereas, we see that VLRs vary and decrease with distance in the other scenarios. During the experiments, we observe that such unlinkage occurs mostly when the vehicles are blocked by buildings, or sometimes blocked by heavy vehicle traffic.

To inspect other factors that may affect VP linkage, we analyze the reception of VDs at both vehicles. Fig. 16 shows a scatter plot of average PDR vs. RSSI obtained from this experiment. It is generally true that higher RSSI results in better PDRs. However, when RSSI values fall in a range of -100 dBm to -80 dBm, we observe the fluctuating PDRs,[11] making it a less likely impacting factor to VP linkage. We also run experiments with different vehicle speed to see whether vehicle mobility such as Doppler effect affects VP linkage (Fig. 17). We see that, in each highway scenario, VLRs are insensitive to velocity for any given separation distance. We rather observe that traffic density on the road affects VP linkage. In our highway experiments, we obtain high(/low) VLRs when the traffic volume is heavy(/light). This result suggests that blockage by heavy vehicle traffic is also likely a impacting factor.

From this set of experiments, we observe that *distance, RSSI, and vehicle mobility have little impact on VLRs, rather line-of-sight condition appears a dominating factor to VP linkage*.

[11] This observation conforms to the previous results reported in [17].

Scenario	Condition	VP linkage	On Video
Open road	LOS	100%	100%
Building 1	NLOS	0%	0%
Intersection 1	LOS	100%	93%
Intersection 2	NLOS	9%	0%
Overpass 1	LOS	84%	77%
Overpass 2	NLOS	0%	0%
Traffic	LOS/NLOS	61%	52%
Vehicle array	NLOS	13%	0%
Pedestrians	LOS	100%	100%
Tunnels	NLOS	0%	0%
Building 2	LOS/NLOS	39%	18%
Double-deck bridge	NLOS	0%	0%
House	LOS/NLOS	56%	51%
Parking structure	NLOS	3%	0%

Table 2: Summary of our measurement results.

7.2.2 Strong Correlation between Linked VPs and Contents of Their Videos

To further examine our observation, we conduct a set of semi-controlled experiments. We carefully select various locations where two vehicles are situated in line-of-sight (LOS) or non line-of-sight (NLOS) or mixed conditions. We measure VLRs to analyze how such conditions affect VP linkage. Table 2 presents the measurement summary from those various scenarios (Pictures of some of our senarios are given in Fig. 19). As shown in the table, our measurements demonstrate that obstacles, especially artificial structures (e.g., buildings and bridges) in vehicular environments cause significant impact on VP linkage.

Given LOS-based VP linkage, we investigate the relationship between linked VPs and their videos. We review the videos taken by the vehicles in the experiments, and discover that, *either vehicle appears on the other's video only when two VPs (of two vehicles) are linked*. The results are reported in Table 2, where 'On Video' indicates

(a) Intersection 1 (LOS). (b) Intersection 2 (NLOS).

(c) Overpass 1 (LOS). (d) Overpass 2 (NLOS).

Figure 19: Pictures of our experiments.

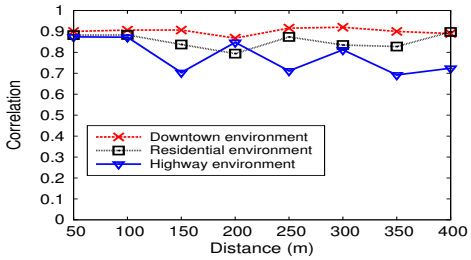


Figure 20: Correlation of VP links and video contents.

the case that either video of two time-aligned VPs captured the other vehicle at least for a moment. We see clear dependency between VP linkage and video contents.

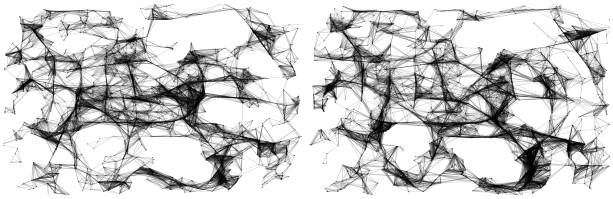
To further validate our analysis, we quantify the degree of association between two events, i.e., linkage between two VPs and visibility on their videos, using the Pearson correlation coefficient [28] as a function of the separation distances across all the data collected from the experiments. The result is shown in Fig. 20, and exhibits a strong degree of association. The correlation varies from 0.7 to 0.9, indicating that VP linkage is indeed associated with the shared "view". This also suggests that when a vehicle involved in a traffic accident has its own VP_u, then videos of the neighbor VPs that are linked to VP_s highly likely captured the accident.

8 Evaluation

To evaluate ViewMap in a large-scale environment, we run simulations using traffic traces.

Simulation setting. We use the network simulator *ns*-3 [29] based on traffic traces of 1000 vehicles obtained from SUMO [30]. We extract a street map (8×8km^2) of the city of Seoul via OpenStreetMap [31]. Output of each simulation is a collection of VPs (each for 1-min). Given those VPs, we construct viewmaps, each of which corresponds to a single 1-min during simulation time.

Simulation results. We first examine the features of such traffic-derived viewmaps. To give a feel for how



(a) Vehicle speed: 50km/h **(b)** Vehicle speed: 70km/h

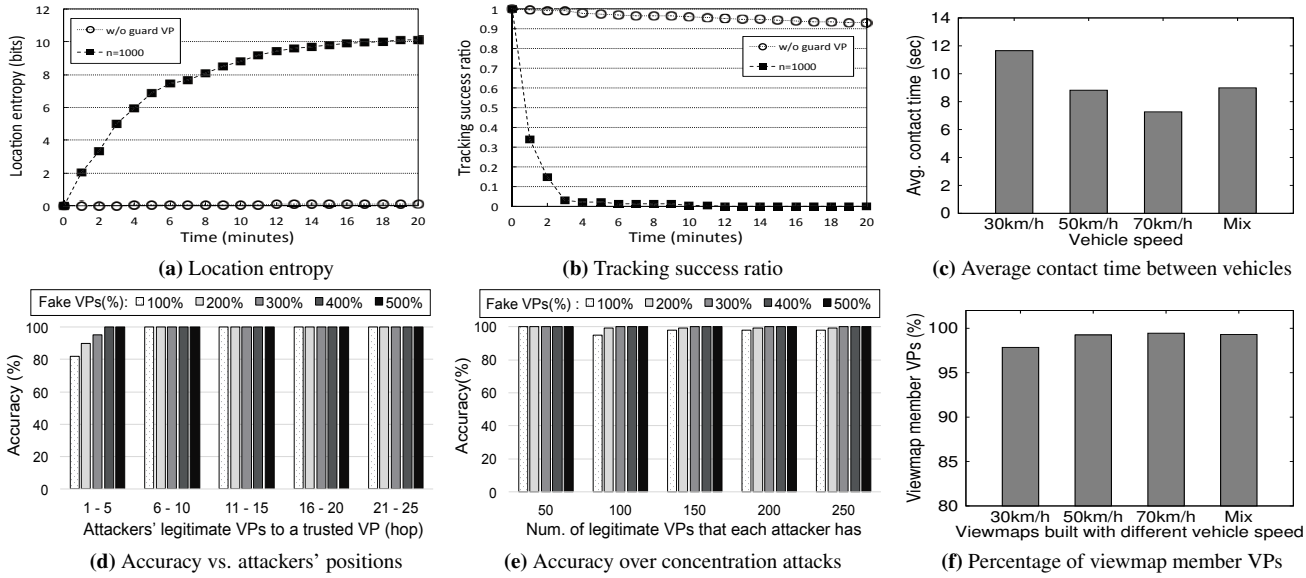
Figure 21: Viewmaps from traffic traces.

they actually look, Fig. 21 depicts viewmaps[12] built from VPs of our simulations where all vehicles move at an average speed of 50km/h (Fig. 21a) and an average speed of 70km/h (Fig. 21b), respectively. Fig. 22c shows the average contact interval between vehicles. We see that contact intervals are not very short in general. This result suggests that vehicles have sufficient time to establish VP links with nearby vehicles as long as they are in line-of-sight condition. Fig. 22f quantifies the viewmaps. We see the some VPs, albeit a few ($<3\%$), are isolated from the viewmaps. We point out that building a viewmap does not require every VP in the area to join. We rather expect that such a case is normal when ViewMap is deployed.

We assess the privacy of the collected VP dataset against tracking as in Section 6. We here present the results of the mix speed scenario as the other cases show similar trends. Fig. 22a shows the average entropy over time. Before ten minutes of driving, vehicles reach eight bits of location entropy, implying that the tracker of a certain vehicle may suspect 256 different locations not knowing exactly where to locate it. Fig. 22b plots the average tracking success ratio over time. It decreases to 0.1 before three minutes and further drops nearly 0.01 before ten minutes. Whereas, it still remains above 0.9 even after twenty minutes without guard VPs. This conforms to the privacy results in a sparser environment given in Section 6 and demonstrates privacy protection in the VP database.

To evaluate the verification performance of the traffic-derived viewmaps, we create fake VPs in the same way as described in Section 6. The results are averaged over 1000 runs of each test set. Fig. 22d plots the verification accuracy in face of fake VPs that outnumber the legitimate ones in the viewmaps. The results confirm the high accuracy—100% in most cases and 82% at worst in the special case of attackers in vicinity of the trusted VP. This is consistent with our analysis in Section 6. We again note that such a case is rare and is a highly restrictive condition for attackers as they cannot predict the future investigation. We also plot the verification accuracy under concentration attacks in Fig. 22e, where each attacker, in a given viewmap, possess as many as 125 legitimate but dummy VPs and create a large number of fake VPs. The result shows that accuracy is still high above 95%. As

[12]The shape of the viewmap reflects the actual road network of a certain area of Seoul that we use for simulations.



(a) Location entropy **(b)** Tracking success ratio **(c)** Average contact time between vehicles

(d) Accuracy vs. attackers' positions **(e)** Accuracy over concentration attacks **(f)** Percentage of viewmap member VPs

Figure 22: Simulation results from different scenarios.

explained, the trust scores of attackers are upperbounded by their topological positions within viewmap's linkage structure (out of their control) rather than their quantity.

9 Related Work

Dashcam video sharing has received little attention so far in the research communities. Recent survey studies [6, 7] find that people with greater privacy concerns have lower reciprocal altruism and justice motive, but have higher monetary motive. These findings have significant implications on the design of dashcam video sharing systems, and ViewMap takes them as design elements.

There have been many proposals on location privacy in a mobile environment, and they focus on hiding users' true positions in the context of location-based services (LBS). *k*-anonymity [32] hides a user's location by using a large region that encloses a group of *k* users in space, instead of a single GPS coordinate. This however decreases spatial accuracy. *CliqueCloak* [33] uses a smaller region but waits for *k* different users in the same region. This delay compromises temporal accuracy. Mix-zones [13] makes users' paths indistinguishable if they coincide in space and time. However, such space-time intersections are not common, especially with frequent and high-accuracy location reports as in our ViewMap. Path confusion [23] resolves the mix-zones' space-time problem by incorporating a delay in revealing users' locations. This delay also compromises temporal accuracy in location data. *CacheCloak* [11] eliminates such delay via users' predicted location reports. Their goal is to hide user identities, but users' claimed positions are not verified.

Existing location verification techniques aim to localize users' positions or to determine the legitimacy of users' location claims. They perform location verification in various ways, from using ultrasound communication [34], to multilateration via infrastructure such as base stations [35, 36] or sensor nodes [37], to directional antenna [38]. Because their objective is to verify users' locations, users are more authenticated rather than anonymized—location privacy is not guaranteed.

10 Conclusion

This paper introduces ViewMap, an automated public service system that enables sharing of dashcam videos while preserving user privacy. The key insight is: (i) to leverage line-of-sight properties of DSRC to link between videos, in their compact form of view profiles (VPs), that share the same sight while driving; and (ii) to exploit inter-vehicle communications to create path confusion for protection against tracking. We use such LOS-based VP links to build a viewmap around a given incident, and identify videos that are worth reviewing. We demonstrate the feasibility of ViewMap via real road experiments. Our evaluation of ViewMap shows high degree of user privacy (tracking success ratio $< 0.1\%$) and high verification accuracy ($> 95\%$). In a broader scope, our solution explores to share private, location-dependent data without resorting to existing infrastructure such as 3G/4G networks where user identities may be exposed.

Acknowledgment

We sincerely thank our shepherd Dr. Ranveer Chandra and the anonymous reviewers for their valuable feedback. This work was supported by Samsung Research Funding Center for Future Technology under Project Number SRFC-IT1402-01.

References

[1] Top Rider. 2015. Research on Vehicle Blackbox. (2015). https://www.top-rider.com:447/news/articleView.html?idxno=19493.

[2] Call made for dashcam rewards. BT. 24 February 2015. http://home.bt.com/lifestyle/motoring/motoring-news/call-made-for-dashcam-rewards-11363964238054.

[3] Police appeal for public to send in dashcam footage of dangerous drivers. Dailymail. 5 January 2014. http://www.dailymail.co.uk/news/article-2534042/.

[4] 2016. "Dashcams P1210466" by Fernost - Own work. Licensed under Public Domain via Commons. (2016).

[5] Shadow Cops in Korea. Daegu Police Department. https://www.dgpolice.go.kr.

[6] Sangkeun Park, Joohyun Kim, Rabeb Mizouni, and Uichin Lee. Motives and concerns of dashcam video sharing. In *Proc. ACM CHI*, 2016.

[7] Kelly Freund. When cameras are rolling: Privacy implications of body-mounted cameras on police. In *Columbia Journal of Law & Social Problems*, 2015.

[8] Tan Zhang, Aakanksha Chowdhery, Paramvir (Victor) Bahl, Kyle Jamieson, and Suman Banerjee. The design and implementation of a wireless video surveillance system. In *Proc. ACM MOBICOM*, 2015.

[9] Marco Gruteser and Xuan Liu. Protecting privacy in continuous location tracking applications. In *IEEE Security and Privacy Magazine*, 2004.

[10] Julien Freudiger, Reza Shokri, and Jean-Pierre Hubaux. Evaluating the privacy risk of location-based services. In *Proc. International Conference on Financial Cryptography and Data Security*, 2011.

[11] Joseph Meyerowitz and Romit Roy Choudhury. Hiding stars with fireworks: Location privacy through camouflage. In *Proc. ACM MOBICOM*, 2009.

[12] Google Directions API. https://developers.google.com/maps/documentation/directions/.

[13] A.R. Beresford and F. Stajano. Location privacy in pervasive computing. In *IEEE Pervasive Computing*, 2003.

[14] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*, 2004.

[15] Zoltán Gyöngyi, Hector Garcia-Molina, and Jan Pedersen. Combating web spam with trustrank. In *Proc. VLDB*, 2004.

[16] David Chaum. Blind signatures for untraceable payments. In *Springer-Verlag*, 1988.

[17] Fan Bai, Daniel D. Stancil, and Hariharan Krishnan. Toward understanding characteristics of dedicated short range communications (DSRC) from a perspective of vehicular network engineers. In *Proc. ACM MOBICOM*, 2010.

[18] OpenCV. http://opencv.org/.

[19] Shyang-Lih Chang, Li-Shien Chen, Yun-Chung Chung, and Sei-Wan Chen. Automatic license plate recognition. 2004.

[20] Duan Tran, Tran Le Hong Du, Tran Vinh Phuoc, and Nguyen Viet Hoang. Building an automatic vehicle license plate recognition system. In *Proc. International Conference in Computer Science*, 2005.

[21] Real-time video processing algorithm for instant license plate recognition in iOS Apps. http://rnd.azoft.com/instant-license-plate-recognition-in-ios-apps/.

[22] Android automatic license plate recognition. https://github.com/SandroMachado/openalpr-android.

[23] Baik Hoh and Marco Gruteser. Preserving privacy in gps traces via uncertainty-aware path cloaking. In *Proc. ACM CCS*, 2007.

[24] Donald B. Reid. An algorithm for tracking multiple targets. 1979.

[25] Marco Gruteser and Baik Hoh. On the anonymity of periodic location samples. In *Proc. Conference on Security in Pervasive Computing*, 2005.

[26] Junghoo Cho and Uri Schonfeld. Rankmass crawler: A crawler with high pagerank coverage guarantee. In *VLDB*, pages 375–386, 2007.

[27] Tor on Android. https://www.torproject.org/docs/android.html.

[28] Stefan Rolewicz. Functional analysis and control theory: Linear systems. In *volume 29 of East European Series. Springer*, 1987.

[29] ns-3. https://www.nsnam.org/.

[30] SUMO. http://sourceforge.net/projects/sumo/.

[31] OpenStreetMap. https://www.openstreetmap.org.

[32] Latanya Sweeney. K-anonymity: A model for protecting privacy. In *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 2002.

[33] B. Gedik and L. Liu. Location privacy in mobile systems: A personalized anonymization model. In *Proc. IEEE ICDCS*, 2005.

[34] Naveen Sastry, Umesh Shankar, and David Wagner. Secure verification of location claims. In *Proc. ACM Workshop on Wireless Security*, 2003.

[35] Srdjan Čapkun, Kasper Bonne Rasmussen, Mario Cagalj, and Mani Srivastava. Secure location verification with hidden and mobile base stations. In *IEEE Transactions on Mobile Computing*, 2008.

[36] Jerry T. Chiang, Jason J. Haas, and Yih-Chun Hu. Secure and precise location verification using distance bounding and simultaneous multilateration. In *Proc. ACM Conference on Wireless Network Security*, 2009.

[37] Srdjan Čapkun and Jean pierre Hubaux. Secure positioning of wireless devices with application to sensor networks. In *Proc. IEEE INFOCOM*, 2005.

[38] Ziwei Ren, Wenfan Li, and Qing Yang. Location verification for vanets routing. In *Proc. WiMob*, 2009.

A Detailed Description of Untraceable Rewarding

The system S posts VP identifier R_u (of reviewed video u) marked as 'request for reward', and corresponding user A obtains the virtual cash that is made untraceable using blind signatures via an anonymous channel as follows:

$$
\begin{aligned}
A \longrightarrow S: &\quad \mathrm{VP}_u, Q_u. \\
S \longrightarrow A: &\quad n \\
A \longrightarrow S: &\quad B(H(m_u^1), r_u^1), ..., B(H(m_u^n), r_u^n) \\
S \longrightarrow A: &\quad \{B(H(m_u^1), r_u^1)\}_{K_S^-}, ..., \{B(H(m_u^n), r_u^b)\}_{K_S^-}
\end{aligned}
$$

where Q_u is a secret number for video u ($R_u = H(Q_u)$) chosen by A when generating VP_u. Upon validation of VP_u and Q_u against u, the system notifies A of the amount of virtual cash (in a predefined monetary unit), n, for video u. Then, A generates n pairs of random messages and their blinding secrets $(m_u^1, r_u^1), ..., (m_u^n, r_u^n)$, and sends the system S their blinded version $B(H(m_u^1), r_u^1), ..., B(H(m_u^n), r_u^n)$ where $B(\cdot)$ is the blinding operation. The system S returns the messages signed with its private key K_S^- without knowing their 'blinded' contents. Then, A unblinds each of the signed messages as follows:

$$U(\{B(H(m_u^i), r_u^i)\}_{K_S^-}) = \{H(m_u^i)\}_{K_S^-}$$

where $U(\cdot)$ is the unblinding operation that requires the blinding secret r_u^i (only known to A). This unblinded signature-message pair $(\{H(m_u^i)\}_{K_S^-}, m_u^i)$ results in one unit of virtual cash. When A presents it for payment, anyone can verify (i) its authenticity via S' signature and (ii) its freshness via double-spending checking on m_u^i, but fails to associate A with his/her video u. Even the system S cannot derive the link between A's virtual cash $(\{H(m_u^i)\}_{K_S^-}, m_u^i)$ and video u's blinded message $B(H(m_u^i), r_u^i)$ without the blinding secret r_u^i, hence untraceable.

B Proof of Lemma 2

Proof. Suppose that v is a fake VP in F_A. Then, the static trust score in for v must be zero because v is clearly not legitimate. By the recursive definition of TrustRank for an individual VP v, we obtain

$$P_v = \alpha \sum_{u \in O_v} \frac{P_u}{|O_u|}. \tag{2}$$

Let E_A be a subset of F_A in which each VP is directly linked by an attacker in A. With an exception for the VPs in E_A, summing Equation (2) for all fake VPs in F_A is the same as we add $\frac{P_u}{|O_u|}$ repeatedly $|O_u|$ times for every $u \in F_A$. Note that for $v \in E_A$, Equation (2) is shown as $P_v = \alpha \sum_{u \in O_v \cap F_A} \frac{P_u}{|O_u|} + \alpha \sum_{u \in O_v \cap A} \frac{P_u}{|O_u|}$. Thus, we get

$$
\begin{aligned}
\sum_{v \in F_A} P_v &= \alpha \sum_{v \in F_A} \sum_{u \in O_v} \frac{P_u}{|O_u|} \\
&= \alpha \left(\sum_{v \in F_A} P_v - \sum_{v \in E_A} \sum_{u \in A} \frac{P_v}{|O_v|} + \sum_{v \in A} \sum_{u \in E_A} \frac{P_v}{|O_v|} \right) \\
&\leq \alpha \sum_{v \in F_A} P_v + \alpha \sum_{v \in A} \frac{|O_v \cap E_A|}{|O_v|} P_v
\end{aligned}
$$

Since $O_v \cap E_A$ is identical to $O_v \cap F_A$ with $v \in A$, it is simple to draw the upper bound of $\sum_{v \in F_A} P_v$ presented in Inequation (1). □

A System to Verify Network Behavior of Known Cryptographic Clients

Andrew Chi Robert A. Cochran Marie Nesfield Michael K. Reiter Cynthia Sturton
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

Numerous exploits of client-server protocols and applications involve modifying clients to behave in ways that untampered clients would not, such as crafting malicious packets. In this paper, we develop a system for verifying in near real-time that a cryptographic client's message sequence is consistent with its known implementation. Moreover, we accomplish this without knowing all of the client-side inputs driving its behavior. Our toolchain for verifying a client's messages explores multiple candidate execution paths in the client concurrently, an innovation useful for aspects of certain cryptographic protocols such as message padding (which will be permitted in TLS 1.3). In addition, our toolchain includes a novel approach to symbolically executing the client software in multiple passes that defers expensive functions until their inputs can be inferred and concretized. We demonstrate client verification on OpenSSL and BoringSSL to show that, e.g., Heartbleed exploits can be detected without Heartbleed-specific filtering and within seconds of the first malicious packet. On legitimate traffic our verification keeps pace with Gmail-shaped workloads, with a median lag of 0.85s.

1 Introduction

Tampering with clients in client-se
plications is an ingredient in nume
abuses can involve exploits on the s
nipulation of application state for w
thoritative. An example of the form
Heartbleed [14] vulnerability, whicl
SSL client to extract contents of se
ample of the latter is an "invalid co
that permits a client greater powers

The ideal defense would be to
verified servers that incorporate all
idation and application-specific c

in practice, current production servers have codebases too large to retrofit into a formally verified model (see Sec. 2). Take for example the continued discovery of critical failures of input validation in all major implementations of Transport Layer Security (TLS) [24]. Despite extensive review, it has been difficult to perfectly implement even "simple" input validation [28], let alone all higher-level program logic that could affect authentication or could compromise the integrity and confidentiality of sensitive data [27].

Since it is generally impossible to anticipate all such abuses, in this paper we explore a holistic approach to validating client behavior as consistent with a sanctioned client program's source code. In this approach, a *behavioral verifier* monitors each client message as it is delivered to the server, to determine whether the sequence of messages received from the client so far is consistent with the program the client is believed to be running and the messages that the server has sent to the client (Fig. 1). Performing this verification is challenging primarily because inputs or nondeterministic events at the client may be unknown to the verifier, and thus, the verifier must de-

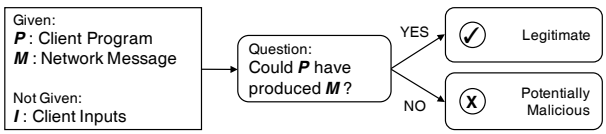


Figure 1: Abstracted behavioral verification problem.

Our central contribution is to show that legitimate cryptographic client behavior can in fact be verified, not against a simplified protocol model but against the client

source code. Intuitively, we limit an attacker to only behaviors that could be effected by a legitimate client. We believe this advance to be important: in showing that messages from a client can be quickly verified as legitimate or potentially malicious, we narrow the time between zero-day exploit and detection to mere seconds. This is significant, since in the case of Heartbleed, for example, the bug was introduced in March 2012 and disclosed in April 2014, a window of vulnerability of two years. During this time, few production networks were even monitoring the relevant TLS Heartbeat records, let alone were configured to detect this misbehavior.

Our examination of 70 OpenSSL CVEs from 2014-2016 showed that 23 out of 37 TLS/DTLS server-side vulnerabilities required tampering with feasible client behavior as an ingredient in exploitation. The vulnerabilities comprised input validation failures, memory errors, data leaks, infinite loops, downgrades, and invalid authorization. Our technique accomplishes verification with no vulnerability-specific configuration and, indeed, could have discovered all of these client exploit attempts even prior to the vulnerabilities' disclosure.

Following several other works in verification of client messages when some client-side values are unknown (see Sec. 2), our strategy is to use symbolic execution [7] to trace the client execution based on the messages received so far from the client (and the messages the server has sent to it). When the verifier, in tracing client execution, operates on a value that it does not know (a "symbolic" value), it considers all possibilities for that value (e.g., branching in both directions if a branch statement involves a symbolic variable) and records constraints on those symbolic values implied by the execution path taken. Upon an execution path reaching a message send point in the client software, the verifier reconciles the accumulated constraints on that execution path with the next message received from the client. If the path does not contradict the message, then the message is confirmed as consistent with some valid client execution.

We advance this body of research in two ways.

1. Prior research on this form of client verification has primarily focused on carefully prioritizing candidate paths through the client in the hopes of finding one quickly to validate the message trace observed so far. This prioritization can itself be somewhat expensive (e.g., involving edit-distance computations on execution paths) and prone to error, in which case the verifier's search costs grow dramatically (e.g., [10]). Here we instead use parallelism to explore candidate paths concurrently, in lieu of sophisticated path prediction. In Sec. 6, we highlight one aspect of cryptographic protocols for which efficiently validating client-side behavior depends on being able to explore multiple execution fragments in parallel, namely execution fragments reflecting plaintexts of different sizes, when the true plaintext size is hidden by message padding (as in SSH and draft TLS 1.3). In this case, predicting the plaintext length is not possible from the ciphertext length, by design, and so exploring different candidate lengths in parallel yields substantial savings.
2. When verifying the behavior of a client in a cryptographic protocol such as TLS, the search for a client execution path to explain the next client message can be stymied by paths that contain cryptographic functions for which some inputs are unknown (i.e., symbolic). The symbolic execution of, e.g., the AES block cipher with an unknown message or a modular exponentiation with an unknown exponent is simply too costly. Every message-dependent branch in the modular exponentiation routine would need to be explored, and the large circuit representation of AES would result in unmanageably complex formulas. In Sec. 5 we thus describe a multi-pass algorithm for exploring such paths, whereby user-specified "prohibitive" functions are bypassed temporarily until their inputs can be deduced through reconciliation with the client message; only then is the function explored (concretely). In cases where those inputs can never be inferred—as would be the case for an ephemeral Diffie-Hellman key, for example—the system outputs the assumption required for the verification of the client message to be correct, which can be discharged from a whitelist of assumptions. Aside from these assumptions, our verification is exact: the verifier accepts if and only if the client is compliant.

Our technique, while not completely turnkey, does not require detailed knowledge of the protocol or application being verified. For example, the specification of prohibitive functions and a matching whitelist of permissible assumptions is straightforward in our examples: the prohibitive functions are simply the AES block cipher, hash functions, and elliptic curve group operations; and the whitelist amounts to the assumption that a particular value sent by the client is in the elliptic-curve group (which the server is required to check [26]). Aside from specifying the prohibitive functions and the whitelist, the other needed steps are identifying network send and receive points, minor syntactic modifications to prepare the client software for symbolic execution (see Appendix B), and, optionally, "stubbing out" calls to software that are irrelevant to the analysis (e.g., `printf`). In the case of validating TLS client behavior, we also leverage a common diagnostic feature on servers: logging session keys to enable analysis of network captures.

We show that client verification can coarsely keep pace with a workload equivalent to an interactive Gmail session running over TLS 1.2, as implemented by

OpenSSL and BoringSSL. Verification averages 49ms per TLS record on a 3.2GHz processor, with 85% completing within 36ms and 95% completing within 362ms. Taking into account the bursts of network activity in Gmail traffic, and that a client-to-server record cannot begin verification until all previous client-to-server records are verified, the median verification *lag* between the receipt of a client-to-server record and its successful verification is 0.85s. We also show that our technique similarly keeps pace with TLS 1.2 connections modified to use message padding, a draft TLS 1.3 [30] feature that introduces costs that our parallel approach overcomes.

Our verifier compares client messages against a specific client implementation, and so it is most appropriate in scenarios where an expected client implementation is known. For example, while a plethora of TLS implementations exist on the open internet, only a few TLS clients are likely to be part of a corporate deployment where installed software is tightly controlled. Knowledge of the implementation might also arise by the client revealing its identification string explicitly (e.g., SSH [38]) or by particulars of its handshake (e.g., TLS [1]). That said, we have also made progress on generalizing our verifier to apply across multiple minor revisions (see Sec. 7).

2 Related Work

The most closely related work is due to Bethea et al. [3] and Cochran and Reiter [10]. These works develop algorithms to verify the behavior of (non-cryptographic) client applications in client-server settings, as we do here. Bethea et al. adopted a wholly offline strategy, owing to the expense of their techniques. Cochran and Reiter improved the method by which a verifier searches for a path through the client program that is consistent with the messages seen by the verifier so far. By leveraging a training phase and using observed messages to provide hints as to the client program paths that likely produced those messages, their technique achieved improved verification latencies but still fell far short of being able to keep pace with, e.g., highly interactive games. Their approach would not work for cryptographic protocols such as those we consider here, since without substantial protocol-specific tuning, the cryptographic protections would obscure information in messages on which their technique depends for generating these hints.

Several other works have sought to verify the behavior of clients in client-server protocols. Most permit false rejections or acceptances since they verify client behavior against an abstract (and so imprecise) model of the client program (e.g., [16, 17]), versus an actual client program as we do here. Others seek exact results as we do, but accomplish this by modifying the client to send all inputs it processes to the verifier, allowing the verifier to simply replay the client on those inputs [34]. In our work, we verify actual client implementations and introduce no additional messaging overhead. Proxies for inferring web-form parameter constraints when a web form is served to a client, to detect parameter-tampering attacks when the form values are returned [32], also provide exact detection. However, this work addresses only stateless clients and does so without attention to cryptographically protected traffic. Our work permits stateful clients and specifically innovates to overcome challenges associated with cryptographic protocols.

Also related to our goals are works focused on verifying the correctness of outsourced computations. Recent examples, surveyed by Walfish and Blumberg [35], permit a verifier to confirm (probabilistically) that an untrusted, remote party performed the outsourced computation correctly, at a cost to the verifier that is smaller than it performing the outsourced computation itself. Since we approach the problem from the opposite viewpoint of a well-resourced verifier (e.g., running with the server in a cloud that the server owner trusts), our techniques do not offer this last property. However, ours requires no changes to the party being verified (in our case, the client), whereas these other works increase the computational cost for the party being verified by orders of magnitude (e.g., see [35, Fig. 5]). Another area of focus in this domain has been reducing the privacy impact of the extra information sent to the verifier to enable verification (e.g., [29]). Since our technique does not require changes to the messaging behavior of the application at all, our technique does not suffer from such drawbacks.

More distantly related is progress on proving security of reference implementations of cryptographic protocols relative to cryptographic assumptions (e.g., miTLS, a reference implementation of TLS in F# [5]) or of modules that can be incorporated into existing implementations to ensure subsets of functionality (e.g., state-machine compliance [4]). Our work instead seeks to prove a property of the *messages* in an interaction, namely that these messages are consistent with a specified client implementation. As such, our techniques show nothing about the intrinsic security of the client (or server) implementation itself; nevertheless, they are helpful in detecting a broad range of common exploit types, as we show here. Our techniques are also immediately applicable to existing production protocol implementations.

3 Background and Goals

A client-server protocol generates messages msg_0, msg_1, ..., some from the client and some sent by the server. Our goal is to construct a *verifier* to validate the client behavior as represented in the message sequence; the server is trusted. We assume that the client is single-threaded

and that the message order reflects the order in which the client sent or received those messages, though neither of these assumptions is fundamental. [1] Our technique is not dependent on a particular location for the verifier, though for the purposes of this paper, we assume it is near the server, acting as a passive network tap.

Borrowing terminology from prior work [10], the task of the verifier is to determine whether there exists an *execution prefix* of the client that is *consistent* with the messages $msg_0, msg_1, \ldots$, as defined below.

Definition 1. Execution Prefix. An execution prefix Π is a sequence of client instructions that begins at the client entry point and follows valid branching behavior in the client program. The sequence may include calls to POSIX `send()` and `recv()`, which are considered "network I/O instructions" and denoted SEND and RECV.

Definition 2. Consistency. An execution prefix Π_n is *consistent* with msg_0, msg_1, ..., msg_n, iff:

- Π_n contains exactly $n+1$ network I/O instructions $\{\gamma_0, ..., \gamma_n\}$, with possibly other instructions.
- $\forall i \in \{0, \ldots, n\}$, $direction(\gamma_i)$ matches the direction of msg_i, where $direction(\text{SEND})$ is client-to-server and $direction(\text{RECV})$ is server-to-client.
- The branches taken in Π_n were possible under the assumption that msg_0, msg_1,..., msg_n were the messages sent and received.

Consistency of Π_n with msg_0, msg_1,..., msg_n requires that the conjunction of all symbolic postconditions at SEND instructions along Π_n be satisfiable, once concretized using contents of messages $msg_0, msg_1, \ldots, msg_n$ sent and received on that path.

The verifier attempts to validate the sequence msg_0, msg_1, ... incrementally, i.e., by verifying the sequence msg_0, msg_1, ..., msg_n starting from an execution prefix Π_{n-1} found to be consistent with msg_0, msg_1, ..., msg_{n-1}, and appending to it an *execution fragment* that yields an execution prefix Π_n consistent with msg_0, msg_1, ..., msg_n. Specifically, an *execution fragment* is a nonempty sequence of client instructions (i) beginning at the client entry point, a SEND, or a RECV in the client software, (ii) ending at a SEND or RECV, and (iii) having no intervening SEND or RECV instructions. If there is no execution fragment that can be appended to Π_{n-1} to produce a Π_n consistent with msg_0, msg_1, ..., msg_n, then the search resumes by *backtracking* to find another execution prefix $\hat{\Pi}_{n-1}$ consistent with msg_0, msg_1, ..., msg_{n-1}, from which the search resumes for an execution fragment to extend it to yield a $\hat{\Pi}_n$ consistent with msg_0, msg_1, ..., msg_n. Only after all such attempts fail can the client behavior be declared invalid.

Determining if a program can output a given value is only semidecidable (recursively enumerable); i.e., while valid client behavior can be declared as such in finite time, invalid behavior cannot, in general. Thus, an "invalid" declaration may require a timeout on the verification process.[2] However, our primary concern in this paper is verifying the behavior of *valid* clients quickly.

4 Parallel Client Verification

As discussed above, upon receipt of message msg_n, the verifier searches for an execution fragment with which to extend execution prefix Π_{n-1} (consistent with msg_0, ..., msg_{n-1}) to create an execution prefix Π_n that is consistent with msg_0, ..., msg_n. Doing so at a pace that keeps up with highly interactive applications remains a challenge (e.g., [10]). We observe, however, that multiple execution fragments can be explored concurrently. This permits multiple worker threads to symbolically execute execution fragments simultaneously, while coordinating their activities through data structures to ensure that they continue to examine new fragments in priority order. In this section, we give an overview of our parallel verification algorithm; this algorithm is detailed in Appendix A.

In this algorithm, a state σ represents a snapshot of execution in a virtual machine, including all constraints (path conditions) and memory objects, which include the contents (symbolic or concrete) of registers, the stack and the heap. We use σ.cons to represent the constraints accumulated during the execution to reach σ, and σ.nxt to represent the next instruction to be executed from σ. The verifier produces state σ_n by symbolically executing the execution prefix Π_n.

The algorithm builds a binary tree of Node objects. Each node nd has a field nd.path to record a path of instructions in the client; a field nd.state that holds a symbolic state; children fields nd.child_0 and nd.child_1 that point to children nodes; and a field nd.saved that will be described in Sec. 5. The tree is rooted with a node nd holding the state $\text{nd.state} = \sigma_{n-1}$ and $\text{nd.path} = \Pi_{n-1}$. The two children of a node nd in the tree extend nd.path through the next symbolic branch (i.e., branch instruction with a symbolic condition). One child node holds a state with a constraint that maintains that the branch condition implies false, and the other child node's state holds a constraint that indicates that the branch condition

[1] The verifier can optimistically assume the order in which it observes the messages is that in which the client sent or received them, which will often suffice to validate a legitimate client even if not strictly true, particularly when the client-server protocol operates in each direction independently (as in TLS). In other cases, the verifier could explore other orders when verification with the observed order fails. Moreover, several works (e.g., [8, 2]) have made progress on symbolic execution of multi-threaded programs.

[2] Nevertheless, our tool declares our tested exploit traces as invalid within several seconds, after exhaustive exploration of the state space. See Sec. 6.1.

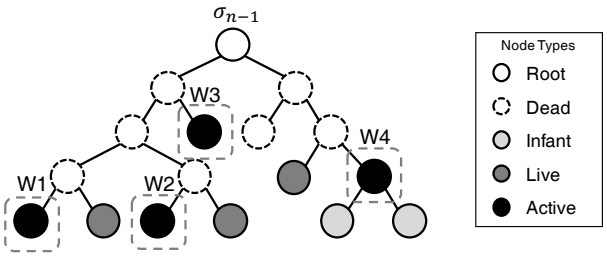


Figure 2: Example node tree.

is true. The algorithm succeeds by finding a fragment with which to extend Π_{n-1} to yield Π_n if, upon extending a path, it encounters a network I/O instruction that yields a state with constraints that do not contradict msg_n being the network I/O instruction's message.

The driving goal of our algorithm is to enable concurrent exploration of multiple states in the node tree. To this end, our algorithm uses multiple threads; one executes a *node scheduler* and the others are *worker* threads, each assigned to one node in the node tree at a time (chosen by the node scheduler, which manages the prioritized heap of unassigned nodes). Fig. 2 shows an example assignment of four workers to multiple nodes in a node tree rooted at σ_{n-1}. White nodes with dashed outlines are *dead* and represent intermediate states that no longer exist. A node is dead if its accumulated constraints reach a contradiction or it has generated children nodes and delivered them to the node scheduler. Black nodes are *active* and are currently being explored by worker threads. Dark-gray nodes are being prioritized by the node scheduler and are still *live*. If there are worker threads that are ready to process a node, they will take their next node from a prioritized queue of the live nodes. Light-gray nodes are *infant* nodes that have just been produced by a worker thread and not yet prioritized by the node scheduler. We can see that worker W4 recently hit a symbolic branch condition and created two infant nodes. The other workers are likely processing straight-line code.

In our design and experiments, the number of worker threads is a fixed parameter provided to the verifier. Because the verification task is largely CPU-bound, in our experience it is not beneficial to use more worker threads than the number of logical CPU cores, and in some cases, fewer worker threads than cores are necessary.

5 Multipass Client Verification

Concurrent exploration of execution fragments can be highly beneficial to the speed of validating legitimate client behavior in cryptographic protocols, as we will show in Sec. 6.3. Nevertheless, there remain challenges to verifying cryptographic clients that no reasonable amount of parallelization can overcome, since doing so would be tantamount to breaking some of the underlying cryptographic primitives. In this section, we introduce a strategy for client verification that can overcome these hurdles for practical protocols such as TLS.

The most obvious challenge is encrypted messages. To make sense of these messages, the verifier needs to be given the symmetric session key under which they are encrypted. Fortunately, existing implementations of, e.g., OpenSSL servers, enable logging session keys to support analysis of network captures, and so we rely on such facilities to provide the session key to the verifier. Given this, it is theoretically straightforward to reverse the encryption on a client-to-server message mid-session—just as the server can—but that capability does surprisingly little to itself aid the verification of the client's behavior, as higher-level protocol logic often composes cryptographic primitives in complex ways. Indeed, state-of-the-art servers routinely fail to detect problems with the message sequence received from a client, as demonstrated by numerous such CVEs [24].

We thus continue with our strategy of incrementally building an execution prefix Π in the client software as each message is received by the verifier to validate the client's behavior. The verifier injects the logged session key into the execution prefix at the point where the key would first be generated by the client. Still, however, the number of execution fragments that need to be explored in cryptographic client implementations is far too large to be overcome by concurrent exploration alone, when other inputs to cryptographic algorithms can be symbolic. Some of these (e.g., a message plaintext, once decrypted) could be injected by the verifier like the session key is, but in our experience, configuring where to inject what values would require much more client-implementation-specific knowledge and bookkeeping than injecting just the session key does. This is in part due to the many layers in which cryptography is applied in modern protocols; e.g., in the TLS handshake, multiple messages are hashed to form the plaintext of another message, which is then encrypted and authenticated. Even worse, other values, e.g., a client's ephemeral Diffie-Hellman key, will never be available to a verifier (or server) and so cannot be injected into an execution prefix.

These observations motivate a design whereby the verifier skips specified functions that would simply be too expensive to execute with symbolic inputs. Specifying such *prohibitive functions* need not require substantial client-implementation-specific or even protocol-specific knowledge; in our experience with TLS, for example, it suffices to specify basic cryptographic primitives such as modular exponentiation, block ciphers, and hash func-

tions as prohibitive. Once specified as prohibitive, the function is skipped by the verifier if any of its inputs are symbolic, producing a symbolic result instead. Once reconciled with the message sequence $msg_0, \ldots, msg_n$ under consideration, however, the verifier can solve for some values that it was previously forced to keep symbolic, after which it can go back and verify function computations (concretely) it had previously skipped. Once additional passes yield no new information, the verifier outputs any unverified function computations (e.g., ones based on the client's ephemeral Diffie-Hellman key) as assumptions on which the verification rests. Only if one of these assumptions is not true will our verifier erroneously accept this message trace. As we will see, these remaining assumptions for a protocol like TLS are minimal.

5.1 User configuration

To be designated prohibitive, a function must meet certain requirements: it must have no side effects other than altering its own parameters (or parameter buffers if passed by reference) and producing a return value; given the same inputs, it must produce the same results; and it must be possible to compute the sizes of all output buffers as a function of the sizes of the input buffers. A function should be specified as prohibitive if it would produce unmanageably complex formulas, such as when the function has a large circuit representation. A function should also be specified as prohibitive if executing it on symbolic inputs induces a large number of symbolic states, due to branching that depends on input values. For example, a physics engine might contain signal processing functions that should be marked prohibitive.

In our case studies, the prohibitive functions are cryptographic functions such as the AES block cipher or SHA-256. We stress, however, that the user need not know how these primitives are composed into a protocol. We illustrate this in Appendix B, where we show the user configuration needed for verifying the OpenSSL client, including the specification of the prohibitive functions. (The configuration for BoringSSL is similar.)

Specifying prohibitive functions generalizes the normal procedure used by symbolic execution to inject symbolic inputs into the program. The user normally designates "user input" functions (such as `getchar`) as symbolic, so that each one is essentially replaced with a function that always returns a symbolic, unconstrained value of the appropriate size. The random number generators, client-side inputs (i.e., `stdin`), and functions that return the current time are also typically so designated. The user configuration for prohibitive functions simply extends this mechanism so that some of these functions do not always return symbolic outputs, but return concrete outputs when their inputs are fully concrete.

5.2 Algorithm overview

The multipass verification algorithm works as follows, when verifying a message msg_n starting from Π_{n-1}. The algorithm expands the binary tree of nodes as described in Sec. 4, with two main differences. First, if the next instruction is a call to a prohibitive function, it is treated as follows: If the prohibitive function is being called with any symbolic input buffers, then its execution is skipped and its outputs are instantiated with fully symbolic output buffers of the appropriate size. If, on the other hand, the prohibitive function is being called with only concrete inputs, then the called function is executed concretely.

Second, upon hitting a network instruction that is consistent with msg_n, the accumulated constraints are saved in a field nd.saved for the node nd that encountered the network instruction. The execution fragment represented by nd is then replayed (starting from Π_{n-1}), again skipping any prohibitive functions encountered with symbolic inputs and concretely executing any encountered with only concrete inputs. Upon hitting the network instruction again, the algorithm compares the previous constraints (saved in nd.saved) with the constraints σ.cons accumulated in the re-execution. If no new constraints have been gathered, then additional re-executions of the execution fragment will similarly gather no new constraints. As such, the execution fragment is appended to Π_{n-1} to create Π_n, since it is consistent with all of $msg_0, \ldots, msg_n$, and the algorithm terminates. Any prohibitive functions that were never concretely executed result in an assumption on which the verification rests—specifically, that there is some input to each such prohibitive function that is consistent with the constraints implied by Π_n and $msg_0, \ldots, msg_n$.

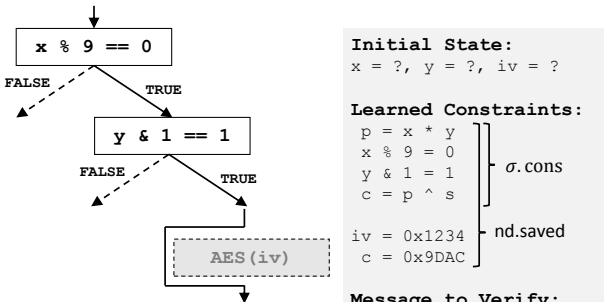
Note that the multipass algorithm for msg_n does not reexamine prohibitive functions that were skipped within Π_{n-1}. In cases where this is desired, lazy constraint generation provides a mechanism to do so, as described in Appendix C.

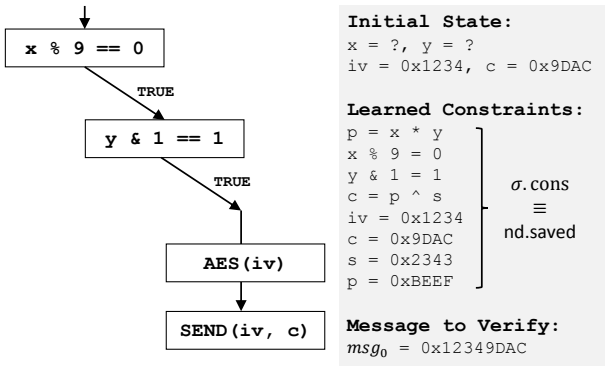
5.3 Detailed walk-through

We now provide a walk-through of this algorithm on a trivial C client shown in Fig. 3a. This client multiplies two of its inputs `x` and `y`, encrypts it using a third input `iv` as an initialization vector, and sends both `iv` and the encrypted value to the server. Our tool begins with a node initialized to the client entry point and attempts to verify (by spawning worker threads) that there exist inputs `x`, `y`, and `iv` that would produce the output message $msg_0 =$ `0x12349DAC` that was observed over the network.

The worker thread that first reaches the SEND has, by that time, accumulated constraints σ.cons as specified in Fig. 3b. Note, however, that it has no constraints relat-

```
void Client(int x, int y, int iv) {
  int p = x*y;
  if (x % 9 == 0) {
    if (y & 1 == 1) {
      int s = AES(iv);
      int c = p ^ s;
      SEND(iv, c);
    }
  }
```





(c) Pass two

Figure 3: Example of multipass verification on a simple client. σ.cons holds the constraints of the execution path and accepted messages leading to the current state.

ing `s` (the output of `AES(iv)`) and `iv`, since `AES` was designated as prohibitive and skipped (since `iv` is symbolic). After reconciling these constraints with the message $msg_0 =$ `0x12349DAC`, the verifier records nd.saved.

The verifier then re-executes from the root (Fig. 3c). Since it now knows `iv = 0x1234`, this time it does not skip `AES` and so computes `s = 0x2343` and `0x9DAC = p ^ 0x2343`, i.e., `p = 0xBEEF`. After this pass, the constraints in nd.saved are still satisfiable (e.g., `x = 0x9, y = 0x1537`). A third pass would add no new information, and so the thread updates the corresponding execution prefix (nd.path) and state (nd.state).

5.4 TLS example

We illustrate the behavior of the multipass algorithm on TLS. Fig. 4 shows an abstracted subset of a TLS client implementation of AES-GCM, running on a single block of plaintext input. For clarity, the example omits details such as the implicit nonce, the server ECDH parameters, the generation of the four symmetric keys, and subsumes the tag computation into the GHASH function. But in all features shown, this walkthrough closely exemplifies the multi-pass verification of a real-world TLS client.

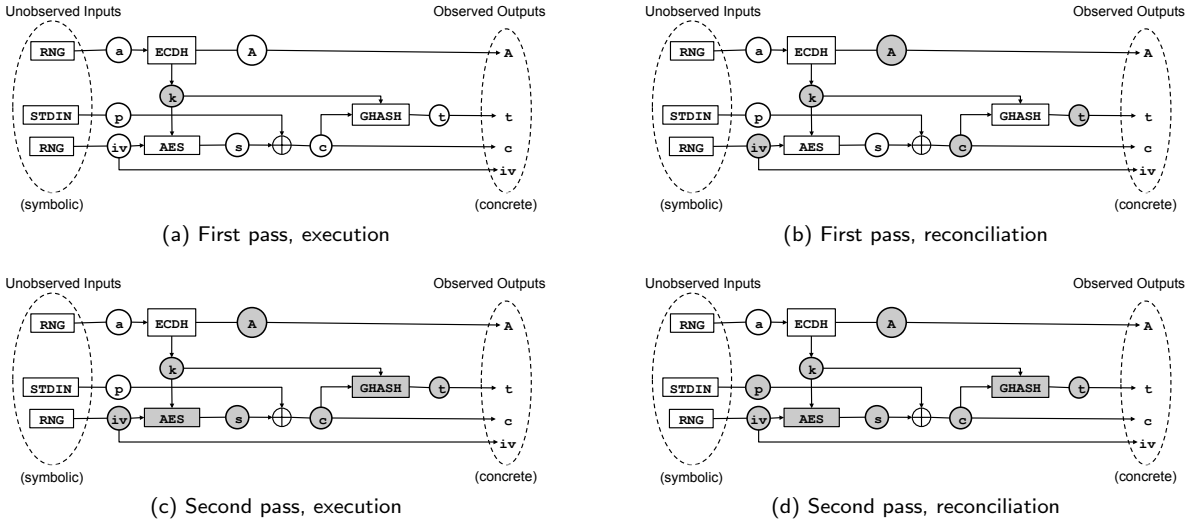
In Fig. 4, the outputs observed by the verifier are the client Diffie-Hellman parameter `A`, the initialization vector `iv`, the ciphertext `c`, and the AES-GCM tag `t`. The unobserved inputs are the Diffie-Hellman private exponent `a`, the initialization vector `iv`, and the plaintext `p`. We do assume access to the AES symmetric key `k`. Since client verification is being performed on the server end of the connection, we can use server state, including `k`.

In the first pass of symbolic execution (Fig. 4a), even with knowledge of the AES symmetric key `k`, all prohibitive functions (ECDH, AES, GHASH) have at least one symbolic input. So, the verifier skips them and produces unconstrained symbolic output for each. After the first execution pass (Fig. 4b), the verifier encounters the observed client outputs. Reconciling them with the accumulated constraints σ.cons yields concrete values for `A`, `t`, `c`, and `iv`, but not the other variables.

The verifier then begins the second pass of symbolic execution (Fig. 4c). Now, AES and GHASH both have concrete inputs and so can be executed concretely. The concrete execution of AES yields a concrete value for `s`, which was not previously known. After the second execution pass (Fig. 4d), the verifier implicitly uses the new knowledge of `s` to check that there is a `p`, the unobserved plaintext value, that satisfies the constraints imposed by observed output. Further passes yield no additional information, as no further symbolic inputs to prohibitive functions can be concretized.

Note that the value of `a`, the client Diffie-Hellman private exponent, is never computed. The verifier thus outputs an assumption that there exists an `a` such that `ECDH(a)` yields values `A` and `k`. As such, we do not detect invalid curve attacks [19], for example; we discuss practical mitigations for this in Sec. 7.3. See Appendix B for the whitelisting of this assumption for a real TLS client.

Note that no decryption mechanism is provided to the verifier. The multipass mechanism automatically recovers the plaintext for stream ciphers and counter-mode block ciphers such as AES-GCM. For other, less preferred modes such as CBC, the user can provide inverse functions via a feature described in Appendix C.



(a) First pass, execution

(b) First pass, reconciliation

(c) Second pass, execution

(d) Second pass, reconciliation

Figure 4: Multipass algorithm on a TLS client implementing an abstracted subset of AES-GCM. Rectangular blocks are prohibitive functions; circles are variables. Shaded nodes are concrete values or functions executed with concrete inputs. Unshaded nodes are symbolic values or skipped functions. In Fig. 4b and Fig. 4d, some values become concrete when σ.cons is reconciled with msg_n.

6 Evaluation

In this section we evaluate our implementation of the algorithms in Secs. 4–5. Our implementation is built on a modified version of KLEE [9], a symbolic execution engine for LLVM assembly instructions, and leverages KLEE's POSIX model, an expanded POSIX model used by Cloud9 [8], and our own model of POSIX network calls. We applied our implementation to verify OpenSSL and BoringSSL clients. BoringSSL, a fork of OpenSSL aiming to improve security and reduce complexity, incorporates changes to the API, thread safety, error handling, and the protocol state machine—resulting in a code base of 200,000 lines of code vs. OpenSSL's 468,000. BoringSSL has been independently maintained since June 2014, and is now deployed throughout Google's production services, Android, and Chrome [23].

Our evaluation goals were as follows. First, we ran a one-worker verifier against two attacks on OpenSSL that represent different classes of client misbehavior, to illustrate detection speed. Second, we load tested a single-worker verifier on a typical TLS 1.2 payload—the traffic generated by a Gmail session—to illustrate the performance of verifying legitimate client behavior. Third, we increased the verification complexity to demonstrate scalability to more complex protocols with larger client state spaces, which we overcome using multiple workers. We did this by verifying a TLS 1.3 draft [30] feature that permits random padding in every packet. The OpenSSL instrumentation (203 lines) and verifier configuration options (138 lines) we used are described in Appendix B; the BoringSSL setup was similar.

The experiments were run on a system with 3.2GHz Intel Xeon E5-2667v3 processors, with peak memory usage of 2.2GB. For the majority of the experiments, a single core ran at 100% utilization. The only exception to this was the third set of experiments (random padding), where up to 16 cores were allocated, though actual utilization varied significantly depending on workload.

Our main performance measure was verification *lag*. To define lag, let the *cost* of msg_n, denoted $cost(n)$, be the wall-clock time that the verifier spends to conclude if msg_n is valid, beginning from an execution prefix Π_{n-1} consistent with $msg_0,\ldots,msg_{n-1}$. That is, $cost(n)$ is the time it spends to produce Π_n from Π_{n-1}. The *completion time* for msg_n is then defined inductively as follows:

$$comp(0) = cost(0)$$
$$comp(n) = \max\{arr(n), comp(n-1)\} + cost(n)$$

where $arr(n)$ is the wall-clock time when msg_n arrived at the verifier. Since the verification of msg_n cannot begin until after both (i) it is received at the verifier (at time $arr(n)$) and (ii) the previous messages $msg_0,\ldots,msg_{n-1}$ have completed verification (at time $comp(n-1)$), $comp(n)$ is calculated as the cost $cost(n)$ incurred after both (i) and (ii) are met. Finally, the *lag* of msg_n is $lag(n) = comp(n) - arr(n)$.

6.1 Misbehavior detection

We first evaluated our client verifier against two attacks on OpenSSL that are illustrative of different classes of

vulnerabilities that we can detect: those related to tampering with the client software to produce messages that a client could not have produced (CVE-2014-0160 Heartbleed) and those with message sequences that, while correctly formatted, would be impossible given a valid client state machine (CVE-2015-0205). Note that testing client *misbehavior* required proof-of-concept attacks, usually prudently omitted from CVEs. We therefore constructed our own attack against each vulnerability and confirmed that each attack successfully exploited an appropriately configured server.

An OpenSSL `1.0.1f s_server` was instantiated with standard settings, and an OpenSSL `s_client` was modified to establish a TLS connection and send a single Heartbleed exploit packet. This packet had a modified length field, and when received by an OpenSSL `1.0.1f s_server`, caused the server to disclose sensitive information from memory. When the trace containing the Heartbleed packet was verified against the original OpenSSL `1.0.1f s_client`, the verifier rejected the packet after exhausting all search paths, with a lag for the Heartbleed packet of 6.9s.

Unlike Heartbleed, CVE-2015-0205 involved only correctly formatted messages. In the certificate exchange, a good client would send a DH certificate (used to generate a pre-master secret), followed by an empty ClientKeyExchange message. A malicious client might send a certificate followed by a ClientKeyExchange message containing a DH parameter. The server would then authenticate the certificate but prefer the second message's DH parameter, allowing a malicious client to impersonate anyone whose public certificate it obtained.

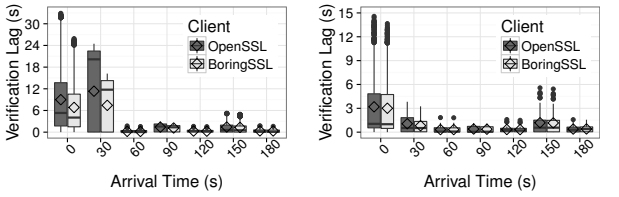
The verifier rejected an attempted attack after a lag of 2.4s, exhausting the search space. This exploit illustrates the power of our technique: we not only verify whether each message is possible in isolation, but also in the context of all previous messages.

Since the tool verifies *valid* client behavior, no attack-specific configuration was required. We do not require any foreknowledge of the exploit and anticipate correct detection of other exploits requiring client tampering.

6.2 Performance evaluation

Our Gmail performance tests measured the lag that resulted from running single-worker verifiers against real-world TLS traffic volumes. The data set was a `tcpdump` capture of a three-minute Gmail session using Firefox, and consisted of 21 concurrent, independent TLS sessions, totaling 3.8MB of network data. This Gmail session was performed in the context of one of the authors' email accounts and included both receiving emails and sending emails with attachments.

In this test we verified the TLS layer of a network connection, but not the application layer above it, such as the browser logic and Gmail web application. To simulate the client-server configuration without access to Gmail servers and private keys, we used the packet sizes and timings from the Gmail `tcpdump` to generate 21 equivalent sessions using OpenSSL `s_client` and `s_server` and the BoringSSL equivalents, such that the amount of traffic sent in each direction at any point in time matched identically with that of the original Gmail capture.[3] The plaintext payload (Gmail web application data) of each session was also replayed exactly, though the payload contents were unlikely to affect TLS performance. One of the 21 TLS sessions was responsible for the vast majority of the data transferred, and almost all of the data it carried was from the server to the client; presumably this was a bulk-transfer connection that was involved in prefetching, attachment uploading, or other latency-insensitive tasks. The other 20 TLS sessions were utilized more lightly and presumably involved more latency-sensitive activities. Since `s_client` implements a few diagnostic features in addition to TLS (but no application layer), verifying `s_client` against these 21 sessions provided a conservative evaluation of the time required to verify the pure TLS layer.



(a) All traffic (b) No server-to-client app traffic

Figure 5: Verification lags for Gmail data set. Box plot at arrival time t includes $\{lag(i) : t \leq arr(i) < t + 30\text{s}\}$. Fig. 5a shows lags for all messages in all 21 TLS sessions. Fig. 5b shows lags if server-to-client application-data messages are dropped.

Fig. 5 shows the distribution of verification lag of messages, grouped by the 30-second interval in which they arrived at the verifier. In each box-and-whisker plot, the three horizontal lines making up each box represent the first, second (median), and third quartiles, and the whiskers extend to cover points within 1.5× the interquartile range. Outliers are shown as single points. In addition, the diamond shows the average value. Fig. 5a show all of the messages' verification lag. It is evident from these figures that the majority of the verification lag

[3] To confirm the appropriateness of using the BoringSSL `s_client`-equivalent in these experiments, we also used it in verification of an unmodified Chrome v50.0.2661.75 browser interacting with an Apache HTTP server.

happened early on, initially up to $\sim$ 30s in the worst case. This lag coincided with an initial burst of traffic related to requests while loading the Gmail application. Another burst occurred later, roughly 160s into the trace, when an attachment was uploaded by the client. Still, the lag for all sessions was near zero in the middle of the session and by the end of the session, meaning that verification for all sessions (in parallel) completed within approximately the wall-clock interval for which the sessions were active.

Verification cost averaged 49ms per TLS record, with 85% costing $\leq$36ms and 95% costing $\leq$362ms. Fig. 6 details cost per message size for all 21 TLS sessions. Despite being smaller, client-to-server messages are costlier to verify, since the verifier's execution of the client software when processing server-to-client messages is almost entirely concrete. In contrast, the execution of the client in preparation of sending a client-to-server message tends to involve more symbolic branching. Also, note the linearity of the relationship between message size and verification cost, particularly for client-to-server messages. This feature suggests a simple, application-independent way to estimate the verification costs for TLS sessions carrying payloads other than Gmail. Assuming similar message sizes in each direction, a deployment could set a sharp timeout at which point the verifier declares the client "invalid." For example, if Fig. 6 were a representative sample of the workloads in a deployment, it would indicate that setting a timeout at a mere 2s (verification cost) could allow the verifier to quickly detect misbehaving clients at a vanishingly small false alarm rate.

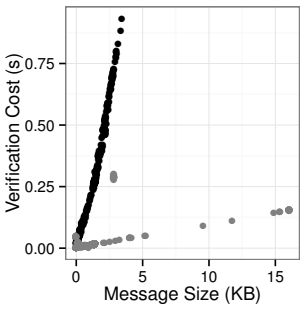


Figure 6: Size versus cost for client-to-server (●) and server-to-client (●) messages.

TLS-Specific Optimizations. While our goal so far had been to provide for client behavior verification with a minimum of protocol-specific tuning, a practical deployment should leverage properties of the protocol for performance. One important property of TLS (and other TCP-based protocols such as SSH) is that its client-to-server and server-to-client message streams operate independently. That is, with the exception of the initial session handshake and ending session teardown, the verifiability of client-to-server messages should be unaffected by which, if any, server-to-client messages the client has received. This gives the verifier the freedom to simply ignore server-to-client application data messages. By doing so, the verification costs for server-to-client messages, now effectively reduced to zero, did not contribute to a growing lag. The effect of this optimization on lag is shown in Fig. 5b, in particular reducing the median lag to 0.85s and the worst-case lag to around 14s. In all subsequent results, we have ignored server-to-client messages unless otherwise noted.

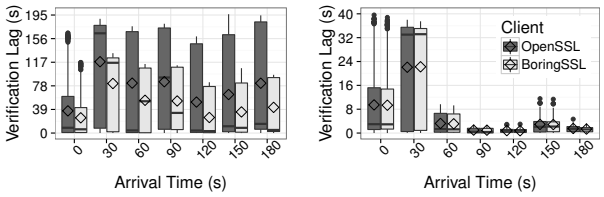
6.3 Stress testing: Added complexity

The Gmail performance evaluation showed that verification of a typical TLS 1.2 session can be done efficiently and reliably, an advance made possible by applying a multipass methodology to cryptographic functions. In essence, once the state explosion from cryptographic functions is mitigated, the client state space becomes small enough that the verification time is primarily determined by the straight-line execution speed of the KLEE symbolic interpreter. However, not all clients are guaranteed to be this simple. One good example is the draft TLS 1.3 standard [30]. In order to hide the length of the plaintext from an observer, implementations of TLS 1.3 are permitted (but not required) to pad an encrypted TLS record by an arbitrary size, up to maximum TLS record size. This random encrypted padding hides the size of the plaintext from any observer, whether an attacker or a verifier. In other words, given a TLS 1.3 record, the length of the input (e.g., from `stdin`) that was used to generate the TLS record could range anywhere from 0 to the record length minus header. Other less extreme examples of padding include CBC mode ciphers, and the SSH protocol, in which a small amount of padding protects the length of the password as well as channel traffic.

We extended our evaluation to stress test our verifier beyond typical current practice. We simulated the TLS 1.3 padding feature by modifying a TLS 1.2 client (henceforth designated "TLS 1.2+"), so that each TLS record includes a random amount of padding up to 128 bytes[4], added before encryption. We then measured verification performance, ignoring server-to-client messages (except during session setup and teardown) as before.

Fig. 7 shows the performance of our single- and 16-worker verifiers on TLS 1.2+ with a random amount of encrypted padding. The addition of random padding to TLS 1.2+ significantly enlarges the client state space that must be explored. With a single-worker verifier, the verification cost increases substantially compared to the TLS 1.2 baseline. The 16-worker verifier reduces the verification cost nearly back to the TLS 1.2 baseline levels. This demonstrates that the state space search is highly amenable to parallelization.

[4]While 128 bytes of padding may seem extreme, previous work showed that an attacker could sometimes infer the website visited by encrypted HTTP connections even with substantial padding (e.g., [25]).



(a) NumWorkers = 1 (b) NumWorkers = 16

Figure 7: Verification lags for Gmail data set when up to 128 bytes of padding are added to each application plaintext, over all 21 TLS sessions. Box plot at arrival time t includes $\{lag(i) : t \leq arr(i) < t+30\text{s}\}$. Fig. 7a shows the lag for a one-worker verifier, and Fig. 7b shows the lag for a 16-worker verifier.

7 Discussion

Here we discuss an approach for dealing with multiple client versions, the applications for which our design is appropriate, and several limitations of our approach.

7.1 Multi-version verifiers

When the version of the client software used by the verifier differs from the version being run by a legitimate client, it is possible for the verifier to falsely accuse the client of being invalid. This poses a challenge for verification when the client version is not immediately evident to the verifier. For example, TLS does not communicate the minor version number of its client code to the server. The possibility for false accusations here is real: we confirmed, e.g., that a verifier for OpenSSL client `1.0.1e` can fail if used to verify traffic for OpenSSL client `1.0.1f`, and vice versa. This occurs because, e.g., the changes from `1.0.1e` to `1.0.1f` included removing MD5 from use and removing a timestamp from a client nonce, among other changes and bug fixes. In total, `1.0.1f` involved changes to 102 files amounting to 1564 insertions and 997 deletions (according to `git`), implemented between Feb 11, 2013 and Jan 6, 2014.

One solution to this problem is to run a verifier for any version that a legitimate client might be using. By running these verifiers in parallel, a message trace can be considered valid as long as one verifier remains accepting of it. Running many verifiers in parallel incurs considerable expense, however.

Another approach is to create one verifier that verifies traffic against several versions simultaneously—a *multi-version verifier*—while amortizing verification costs for their common code paths across all versions. To show the potential savings, we built a multi-version verifier for both `1.0.1e` and `1.0.1f` by manually assembling a "unioned client" of these versions, say "`1.0.1ef`". In client `1.0.1ef`, every difference in the code between client `1.0.1e` and client `1.0.1f` is preceded by a branch on version number, i.e.,

```
if (strcmp(version, "1.0.1e") == 0) {
   /* 1.0.1e code here */
} else {
   /* 1.0.1f code here */
}
```

We then provided this as the client code to the verifier, marking `version` as symbolic. Note that once the client messages reveal behavior that is consistent with only one of `1.0.1e` and `1.0.1f`, then `version` will become concrete, causing the verifier to explore only the code paths for that version; as such, the verifier still allows only "pure `1.0.1e`" or "pure `1.0.1f`" behavior, not a combination thereof.

The single-worker costs (specifically, $\sum_i cost(i)$) of verifying `1.0.1e` traffic with a `1.0.1ef` verifier and of verifying `1.0.1f` traffic with a `1.0.1ef` verifier were both within 4% of the costs for verifying with a `1.0.1e` and `1.0.1f` verifier, respectively. (For these tests, we used the same Gmail traces used in Sec. 6.) Despite a 32% increase in symbolic branches and a 7% increase in SMT solver queries, the overall cost increases very little. This implies that despite an increase in the number of code path "options" comprising the union of two versions of client code, the incorrect paths die off quickly and contribute relatively little to total verification cost, which is dominated by straight-line symbolic execution of paths common to both versions.

While a demonstration of a multi-version verifier for only two versions of one codebase, we believe this result suggests a path forward for verifying clients of unknown versions much more efficiently than simply running a separate verifier for each possibility. We also anticipate that multi-version verifiers can be built automatically from commit logs to repositories, a possibility that we hope to explore in future work.

7.2 Applications

Suitable Application Layers. Consider a deployment of behavioral verification as an intrusion detection system (IDS). Verification lag determines the period that a server does not know a message's validity. The application layer chosen for our TLS evaluation, Gmail, exhibited relatively high lag due to its high-volume data transfers. Other applications may be more optimal for behavioral verification. For example, XMPP [31] generally sends small XML payloads for text-based Internet messaging. Another setting is electronic mail (SMTP) [21], which originally lacked security. Gradually [15], the internet community has deployed mecha-

nisms such STARTTLS [18], SPF [20], DKIM [12], and DMARC [22], many with cryptographic guarantees of authenticity. Behavioral verification can provide a strong compliance check of these mechanisms. Although data volumes can be large, the application is tolerant of delay, making verification lag acceptable.

Other Cryptographic Protocols. Perhaps due to its use in various applications, TLS is one of the more complex security protocols. We believe that the client verification technique should generalize to other, often simpler, protocols. One example is Secure Shell (SSH) [37, 38]. When used as a remote shell, SSH requires low latency but transfers a relatively small amount of data: key presses and terminal updates. When used for file transfer (SFTP), a large volume of data is sent, but in a mode that is relatively latency-insensitive.

7.3 Limitations

Source Code and Configuration. Our verifier requires the client source code to generate LLVM bitcode and to designate prohibitive functions. We also require knowledge of the client configuration, such as command line parameters controlling the menu of possible cipher suites. Again, our approach is most suitable for environments with a known client and configuration.

Environment Modeling. While OpenSSL `s_client` has relatively few interactions with the environment, other clients may interact with the environment extensively. For example, SSH reads `/etc/passwd`, the `.ssh/` directory, redirects standard file descriptors, etc. The KLEE [9] and Cloud9 [8] POSIX runtimes serve as good starting points, but some environment modeling is likely to be necessary for each new type of client. This one-time procedure per client is probably unavoidable.

Manual Choice of Prohibitive Functions. We currently choose prohibitive functions manually. The choice of hash functions, public key algorithms, and symmetric ciphers may be relatively obvious to security researchers, but not necessarily to a typical software developer.

Prohibitive Function Assumptions. When prohibitive functions are initially skipped but eventually executed concretely, verification soundness is preserved. If a prohibitive function is never executed concretely (e.g., due to asymmetric cryptography), this introduces an assumption; e.g., in the case of ECDH, a violation of this assumption could yield an invalid curve attack [19]. In a practical deployment, the user designating a prohibitive function should also designate predicates on the function's output (e.g., the public key is actually a group element) that are specified by the relevant NIST or IETF standards as mandatory server-side checks [26] (which would have prevented the Jager et al. attack [19]). In our tool, these predicates could be implemented via lazy constraint generation (see Appendix C), or as a `klee_assume` for simple predicates. We recommend typical precautions [13] to avoid Bleichenbacher-type attacks [6].

Denial of Service. We anticipate our verifier being deployed as an IDS via a passive network tap. To mitigate a potential denial of service (DoS) attack, one could leverage the linear relationship between verification cost and message size: (1) Impose a hard upper bound on verifier time per packet, and declare all packets that exceed the time budget invalid. Since our results show legitimate packets finish within a few seconds, the bound could easily be set such that the false alarm rate is negligible. (2) Given a fixed CPU time budget, precisely compute the amount of traffic that can be verified. The operator could then allocate verifiers according to the threat profile, e.g., assigning verifiers to high-priority TLS sessions or ones from networks with poor reputation (e.g., [11]). This would degrade verification gracefully as total traffic bandwidth grows beyond the verification budget.

8 Conclusion

We showed that it is possible to practically verify that the messaging behavior of an untrusted cryptographic client is consistent with its known implementation. Our technical contributions are twofold. First, we built a parallel verification engine that supports concurrent exploration of paths in the client software to explain a sequence of observed messages. This innovation is both generally useful for client verification and specifically useful for verifying cryptographic clients, e.g., due to ambiguities arising from message padding hidden by encryption. Second, we developed a multipass verification strategy that enables verification of clients whose code contains cryptographic functions, which typically pose major challenges to symbolic execution. We demonstrated that our verifier detects two classes of client misbehavior: those that produce malformed messages, and those whose message sequence is impossible. In addition, we showed that our verifier can coarsely keep pace with a Gmail TLS workload, running over both OpenSSL and BoringSSL TLS 1.2 and over a more complex simulation of TLS 1.3. We believe our system could dramatically reduce the detection time of protocol exploits, with no prior knowledge of the vulnerabilities.

Acknowledgements. We thank our shepherd, Boon Thau Loo, and the anonymous reviewers for their comments. This work was supported in part by NSF grants 1115948 and 1330599, grant N00014-13-1-0048 from the Office of Naval Research, and a gift from Cisco.

References

[1] B. Anderson, S. Paul, and D. A. McGrew. Deciphering malware's use of TLS (without decryption). *arXiv preprint*, abs/1607.01639, 2016.

[2] T. Bergan, D. Grossman, and L. Ceze. Symbolic Execution of Multithreaded Programs from Arbitrary Program Contexts. In *2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 491–506, 2014.

[3] D. Bethea, R. A. Cochran, and M. K. Reiter. Server-side verification of client behavior in online games. *ACM Transactions on Information and System Security*, 14(4), Dec. 2011.

[4] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, P.-Y. S. A. Pironti, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *36th IEEE Symposium on Security and Privacy*, pages 535–552, 2015.

[5] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with verified cryptographic security. In *34th IEEE Symposium on Security and Privacy*, pages 445–459, 2013.

[6] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology – CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*. 1998.

[7] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT – a formal system for testing and debugging programs by symbolic execution. In *International Conference on Reliable Software*, pages 234–245, 1975.

[8] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *6th European Conference on Computer Systems*, 2011.

[9] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2008.

[10] R. A. Cochran and M. K. Reiter. Toward online verification of client behavior in distributed applications. In *20th ISOC Network and Distributed System Security Symposium*, 2013.

[11] M. P. Collins, T. J. Shimeall, S. Faber, J. Janies, R. Weaver, M. De Shon, and J. Kadane. Using uncleanliness to predict future botnet addresses. In *7th Internet Measurement Conference*, pages 93–104, 2007.

[12] D. Crocker, T. Hansen, and M. Kucherawy. DomainKeys Identified Mail (DKIM) Signatures. RFC 6376 (Internet Standard), Sept. 2011.

[13] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) protocol version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176, 7465, 7507.

[14] K. Durumeric, J. Kasten, D. Adrian, A. J. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of Heartbleed. In *Internet Measurement Conference*, 2014.

[15] Z. Durumeric, D. Adrian, A. Mirian, J. Kasten, E. Bursztein, N. Lidzborski, K. Thomas, V. Eranti, M. Bailey, and J. A. Halderman. Neither snow nor rain nor MITM...: An empirical analysis of email delivery security. In *2015 ACM Internet Measurement Conference*, pages 27–39, 2015.

[16] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *11th USENIX Security Symposium*, Aug. 2002.

[17] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *18th International World Wide Web Conference*, pages 561–570, Apr. 2009.

[18] P. Hoffman. SMTP Service Extension for Secure SMTP over Transport Layer Security. RFC 3207 (Proposed Standard), Feb. 2002. Updated by RFC 7817.

[19] T. Jager, J. Schwenk, and J. Somorovsky. Practical invalid curve attacks on TLS-ECDH. In *Computer Security – ESORICS 2015*, volume 9326 of *Lecture Notes in Computer Science*. 2015.

[20] S. Kitterman. Sender Policy Framework (SPF) for authorizing use of domains in email, version 1. RFC 7208 (Proposed Standard), Apr. 2014. Updated by RFC 7372.

[21] J. Klensin. Simple Mail Transfer Protocol. RFC 5321 (Draft Standard), Oct. 2008. Updated by RFC 7504.

[22] M. Kucherawy and E. Zwicky. Domain-based Message Authentication, Reporting, and Conformance (DMARC). RFC 7489 (Informational), Mar. 2015.

[23] A. Langley. BoringSSL. ImperialViolet, Oct. 2015. https://www.imperialviolet.org/2015/10/17/boringssl.html.

[24] J. Leyden. Annus HORRIBILIS for TLS! all the bigguns now officially pwned in 2014. The Register, Nov. 2014. http://www.theregister.co.uk/2014/11/12/ms_crypto_library_megaflaw/.

[25] M. Liberatore and B. N. Levine. Inferring the source of encrypted HTTP connections. In *13th ACM Conference on Computer and Communications Security*, pages 255–263, 2006.

[26] D. McGrew, K. Igoe, and M. Salter. Fundamental elliptic curve cryptography algorithms. RFC 6090 (Proposed Standard), Feb. 2011.

[27] MITRE. Divide-and-conquer session key recovery in SSLv2 (OpenSSL). CVE-2016-0703, Mar. 1 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0703.

[28] MITRE. Memory corruption in the ASN.1 encoder (OpenSSL). CVE-2016-2108, May 3 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2108.

[29] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, May 2013.

[30] E. Rescorla. The Transport Layer Security (TLS) protocol version 1.3. Internet-Draft draft-ietf-tls-tls13-18 (work in progress), IETF Secretariat, October 2016.

[31] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): core. RFC 6120 (Proposed Standard), Mar. 2011.

[32] N. Skrupsky, P. Bisht, T. Hinrichs, V. N. Venkatakrishnan, and L. Zuck. TamperProof: A server-agnostic defense for parameter-tampering attacks on web applicatoins. In *3rd ACM Conference on Data and Application Security and Privacy*, Feb. 2013.

[33] S. Vaudenay. Security flaws induced by CBC padding - Applications to SSL, IPSEC, WTLS... In *Advances in Cryptology - EUROCRYPT 2002*, pages 534–546, 2002.

[34] K. Vikram, A. Prateek, and B. Livshits. Ripley: Automatically securing Web 2.0 applications through replicated execution. In *16th ACM Conference on Computer and Communications Security*, Nov. 2009.

[35] M. Walfish and A. J. Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2), Feb. 2015.

[36] S. Webb and S. Soh. A survey on network game cheats and P2P solutions. *Australian Journal of Intelligent Information Processing Systems*, 9(4):34–43, 2008.

[37] T. Ylonen and C. Lonvick. The Secure Shell (SSH) authentication protocol. RFC 4252 (Proposed Standard), Jan. 2006.

[38] T. Ylonen and C. Lonvick. The Secure Shell (SSH) transport layer protocol. RFC 4253 (Proposed Standard), Jan. 2006. Updated by RFC 6668.

A Algorithm Details

The algorithm for verifying a client-to-server message works as follows. This algorithm, denoted ParallelVerify, takes as input the execution prefix Π_{n-1} consistent with $msg_0, \ldots, msg_{n-1}$; the symbolic state σ_{n-1} resulting from execution of Π_{n-1} from the client entry point on message trace $msg_0, \ldots, msg_{n-1}$; and the next message msg_n. Its output is Rslt, which holds the prefix Π_n and corresponding state σ_n in Rslt.path and Rslt.state, respectively, if a prefix consistent with $msg_0, \ldots, msg_n$ is found. If the procedure returns with $\mathsf{Rslt.path} = \mathsf{Rslt.state} = \perp$, then this indicates that there is no execution prefix that can extend Π_{n-1} to make Π_n that is consistent with msg_0, …, msg_n. This will induce backtracking to search for another $\hat{\Pi}_{n-1}$ that is consistent with msg_0, …, msg_{n-1}, which the verifier will then try to extend to find a $\hat{\Pi}_n$ consistent with msg_0, …, msg_n.

A.1 Parallel verification

ParallelVerify runs in a thread that spawns NumWorkers + 1 child threads: one thread to manage scheduling of nodes for execution via the procedure NodeScheduler (not shown) and NumWorkers worker threads to explore candidate execution fragments via the procedure VfyMsg (Fig. 8).

NodeScheduler manages the selection of node states to execute next and maintains the flow of nodes between worker threads. It receives as input two queues of nodes, a "ready" queue Q_R and an "added" queue Q_A. These queues are shared between the worker threads and the NodeScheduler thread. Worker threads pull nodes from Q_R and push new nodes onto Q_A. As there is only one

scheduler thread and one or more worker threads producing and consuming nodes from the queues Q_R and Q_A, Q_R is a single-producer-multi-consumer priority queue and Q_A is a multi-producer-single-consumer queue. The goal of NodeScheduler is to keep Q_A empty and Q_R full. Nodes are in one of four possible states, either actively being explored inside VfyMsg, stored in Q_R, stored in Q_A, or being prioritized by NodeScheduler. A node at the front of Q_R is the highest priority node not currently being explored. The nodes in Q_A are child nodes that have been created by VfyMsg threads that need to be prioritized by NodeScheduler and inserted into Q_R. NodeScheduler continues executing until the boolean Done is set to true by some VfyMsg thread.

Shown in Fig. 8, the procedure VfyMsg does the main work of client verification: stepping execution forward in the state σ of each node. In this figure, lines shaded gray will be explained in Sec. A.2 and can be ignored for now (i.e., read Fig. 8 as if these lines simply do not exist). VfyMsg runs inside of a while loop until the value of Done is no longer equal to **false** (101). Recall that the parent procedure ParallelVerify spawns multiple instances of VfyMsg. Whenever there is a node on the queue Q_R, the condition on line 102 will be true and the procedure calls dequeue atomically. Note that even if $|Q_R| = 1$, multiple instances of VfyMsg may call dequeue in 103, but only one will return a node; the rest will retrieve undefined ($\perp$) from dequeue.

If nd is not undefined (104), the algorithm executes the state nd.state and extends the associated path nd.path up to either the next network instruction (SEND or RECV) or the next symbolic branch (a branch instruction that is conditioned on a symbolic variable). The first case, stepping execution on a non-network / non-symbolic-branch instruction σ.nxt (here denoted isNormal(σ.nxt)), executes in a while loop on lines 106–108. The current instruction is appended to the path and the procedure execStep is called, which symbolically executes the next instruction in state σ. These lines are where most of the computation work is done by the verifier. Concurrently stepping execution on multiple states is where the largest performance benefits of parallelization are achieved. Note that calls to execStep may invoke branch instructions, but these are non-symbolic branches.

In the second case, if the next instruction is SEND or RECV and if the constraints σ.cons accumulated so far with the symbolic state σ do not contradict the possibility that the network I/O message σ.nxt.msg in the next instruction σ.nxt is msg_n (i.e., $(\sigma.\text{cons} \wedge \sigma.\text{nxt.msg} = msg_n) \not\Rightarrow$ **false**, line 110), then the algorithm has successfully reached an execution prefix Π_n consistent with $msg_0, \ldots, msg_n$. The algorithm sets the termination value (Done = **true**) and sets the return values of the parent function on lines 112–113: Rslt.path is set to

```
100 procedure VfyMsg(msg_n, Root, Q_R, Q_A, Done, Rslt)
101   while ¬Done do
102     if |Q_R| > 0 then
103       nd ← dequeue(Q_R)
104       if nd ≠ ⊥ then
105         π ← nd.path ; σ ← nd.state
106         while isNormal(σ.nxt) do
107           π ← π ‖ ⟨σ.nxt⟩
108           σ ← execStep(σ)
109         if isNetInstr(σ.nxt) then
110           if (σ.cons ∧ σ.nxt.msg = msg_n) ⇏ false then
111             if (σ.cons ∧ σ.nxt.msg = msg_n) ≡ nd.saved then
112               Rslt.path ← π ‖ ⟨σ.nxt⟩
113               Rslt.state ← [execStep(σ) | σ.nxt.msg ↦ msg_n]
114               Done ← true                                  ▷ Success!
115             else
116               nd ← clone(Root)
117               nd.saved ← σ.cons ∧ σ.nxt.msg = msg_n
118               enqueue(Q_A, nd)
119         else if isProhibitive(σ.nxt) then
120           nd.path ← π ‖ ⟨σ.nxt⟩
121           nd.state ← execStepProhibitive(σ, nd.saved)
122           enqueue(Q_A, nd)
123         else if isSymbolicBranch(σ.nxt) then
124           π ← π ‖ ⟨σ.nxt⟩
125           σ′ ← clone(σ)
126           σ′ ← [execStep(σ′) | σ′.nxt.cond ↦ false]
127           if σ′.cons ⇏ false then
128             nd.child_0 ← makeNode(π, σ′, nd.saved)
129             enqueue(Q_A, nd.child_0)
130           σ ← [execStep(σ) | σ.nxt.cond ↦ true]
131           if σ.cons ⇏ false then
132             nd.child_1 ← makeNode(π, σ, nd.saved)
133             enqueue(Q_A, nd.child_1)
```

Figure 8: VfyMsg procedure, described in Appendix A.1. Shaded lines implement the multipass algorithm and are described in Appendix A.2.

the newly found execution prefix Π_n; Rslt.state is set to the state that results from executing it, conditioned on the last message being msg_n (denoted [execStep(σ) | σ.nxt.msg $\mapsto msg_n$]); and any prohibitive functions that were skipped are recorded for outputting assumptions (not shown, for notational simplicity). All other threads of execution now exit because Done = **true** and the parent procedure ParallelVerify will return Rslt.

In the final case, (isSymbolicBranch(σ.nxt)), the algorithm is at a symbolic branch. Thus, the branch condition contains symbolic variables and cannot be evaluated as true or false in isolation. Using symbolic execution, the algorithm evaluates both the true branch and the false branch by executing σ.nxt conditioned on the condition evaluating to **false** (denoted [execStep(σ') | σ'.nxt.cond $\mapsto$ **false**] in line 126) and conditioned on the branch condition evaluating to **true** (130). In each case, the constraints of the resulting state are checked for consistency (127, 131), for example, using an SMT solver.

If either state is consistent, it is atomically placed onto Q_A (129, 133).

A.2 The multipass algorithm

The multipass verification algorithm involves changes to the VfyMsg procedure in Fig. 8, specifically the insertion of the shaded lines. Whenever σ.nxt is a call to a prohibitive function, it is treated separately (lines 119–122), using the execStepProhibitive function (121). (To accomplish this, isNormal in line 106 now returns **false** not only for any network instruction or symbolic branch, but also for any call to a prohibitive function.) If execStepProhibitive receives a call σ.nxt to a prohibitive function with any symbolic input buffers, it replaces the call with an operation producing fully symbolic output buffers of the appropriate size. However, if the constraints saved in nd.saved allow the concrete input buffer values to be inferred, then execStepProhibitive instead performs the call σ.nxt on the now-concrete input buffers.

Prior to the execution path reaching a network instruction, when a call σ.nxt to a prohibitive function is encountered, nd.saved is simply **true** as initialized (not shown), permitting no additional inferences about the values of input buffers to σ.nxt. After a network instruction is reached and msg_n is reconciled with the constraints σ.cons accumulated along the path so far (110), the path constraints σ.cons and the new constraint σ.nxt.msg $= msg_n$ are saved in nd.saved (117). The execution path is then replayed from the root of the binary tree (i.e., beginning from Π_{n-1}, see 116). This process repeats until an execution occurs in which nothing new is learned (i.e., $(\sigma.\text{cons} \wedge \sigma.\text{nxt.msg} = msg_n) \equiv \text{nd.saved}$, in 111), at which point VfyMsg returns as before.

B TLS Experimental Setup

In Sec. 6, we applied our client verification algorithm to OpenSSL, a widely used implementation of Transport Layer Security (TLS) with over 400,000 lines of code. In order to run OpenSSL symbolically in KLEE, some initial instrumentation (203 modified/added lines of code) was required: compiling to LLVM without x86 assembly, inserting symbolics at random number generators, and providing convenient record/playback functionality for testing the network SEND and RECV points. This manual one-time cost is likely unavoidable for symbolic execution of any application, but should be relatively small (0.05% of the OpenSSL codebase).

We then configured our OpenSSL client with one of the currently preferred cipher suites, namely `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256`.

- Key exchange: Ephemeral Elliptic Curve Diffie-Hellman (ECDHE) signed using the Elliptic Curve Digital Signature Algorithm (ECDSA)
- Symmetric Encryption: 128-bit Advanced Encryption Standard (AES) in Galois/Counter Mode
- Pseudorandom function (PRF) built on SHA-256

Since our goal was to verify the TLS layer and not the higher-layer application, in our experiment we took advantage of the OpenSSL `s_client` test endpoint. This client establishes a fully functional TLS session, but allows arbitrary application-layer data to be sent and received via `stdin` and `stdout`, similar to the `netcat` tool. Verifying that network traffic is consistent with `s_client` is roughly equivalent to verifying the TLS layer alone.

The OpenSSL-specific user configuration for verification consisted of the following:

1. Configuring the following OpenSSL functions as prohibitive: `AES_encrypt, ECDH_compute_key, EC_POINT_point2oct, EC_KEY_generate_key, SHA1_Update, SHA1_Final, SHA256_Update, SHA256_Final, gcm_gmult_4bit, gcm_ghash_4bit`
2. Configuring `tls1_generate_master_secret` as the function to be replaced by server-side computation of the symmetric key.
3. (Optional) Declaring `EVP_PKEY_verify` to be a function that always returns success. This is a performance optimization only.
4. Configuring the whitelist of assumptions, containing one element to accept the client's ephemeral Diffie-Hellman public key.

The user configuration for OpenSSL, comprising declarations of prohibitive functions and their respective input/output annotations, consisted of 138 lines of C code using our API, which is implemented using C preprocessor macros. Fig. 9 shows an example prohibitive function declaration for the AES block cipher. In this macro, we declare the function signature, which comprises the 128-bit input buffer `in`, the 128-bit output buffer `out`, and the symmetric key data structure, `key`, which contains the expanded round keys. Both `in` and `key` are checked for symbolic data. If either buffer contains symbolic data, `out` is populated with unconstrained symbolic data, and the macro returns without executing any subsequent lines. Otherwise, the underlying (concrete) AES block cipher is called.

In a pure functional language or an ideal, strongly typed language, the prohibitive function specifications could in principle be generated automatically from the

```
DEFINE_MODEL(void, AES_encrypt,
             const unsigned char *in,
             unsigned char *out,
             const AES_KEY *key)
{
  SYMBOLIC_CHECK_AND_RETURN(
      in, 16,
      out, 16, "AESBlock");
  SYMBOLIC_CHECK_AND_RETURN(
      key, sizeof(AES_KEY),
      out, 16, "AESBlock");
  CALL_UNDERLYING(AES_encrypt,
      in, out, key);
}
```

Figure 9: Example prohibitive function declaration.

function name alone. Unfortunately, in C, the memory regions representing input and output may be accessible only through pointer dereferences and type casts. This is certainly true of OpenSSL (e.g., there is no guarantee that the `AES_KEY` struct does not contain pointers to auxiliary structs). Therefore, for each prohibitive function, the user annotation must explicitly define the data layout of the input and output.

The final configuration step involves specifying a whitelist of permitted assumptions. In the case of OpenSSL, only one is necessary, namely, that the Elliptic Curve Diffie-Hellman (ECDH) public key received from the client is indeed a member of the elliptic curve group (which, incidentally, the server is required to check [26]). In an actual verification run, the verifier produces an assumption of the form:

```
ARRAY(65) ECpointX_0___ECpoint2oct_0 =
  0x04001cfee250f62053f7ea555ce3d8...
```

This assumption can be interpreted as the following statement: the verifier assumes it is possible for the client to create a buffer of length 65 bytes with contents `0x04001cf...` by first making the zeroth call to an ECpoint generation function, and then passing its output through the zeroth call to an ECpoint2oct (serialization) function. The single-element whitelist for OpenSSL is therefore simply:

```
ARRAY(*) ECpointX_*___ECpoint2oct_*
```

The asterisks (`*`) are wildcard symbols; the ECDH public key can be of variable size, and for all n, m, we permit the nth call to `ECpointX` and the mth call to the `ECpoint2oct` functions to be used as part of an entry in the whitelist.

The domain knowledge required for the first two configuration steps is minimal, namely that current TLS configurations use the above cryptographic primitives in some way, and that a symmetric key is generated in a particular function. The domain knowledge necessary for the third configuration step is that TLS typically uses public key signatures only to authenticate the server to the client, e.g., via the Web PKI. The server itself generates the signature that can be verified via PKI, and so the verifier knows that the chain of signature verifications is guaranteed to succeed. Moreover, this optimization generalizes to any protocol that uses a PKI to authenticate the server to an untrusted client. The domain knowledge necessary for the fourth configuration step is non-trivial; whitelisting a cryptographic assumption can have subtle but significant effects on the guarantees of the behavioral verifier (e.g., off-curve attacks). However, the number of assumptions is small: for OpenSSL, there is only the single assumption above.

C Lazy Constraint Generators

There are a number of cases in which a behavioral verifier requires an extra feature, called a *lazy constraint generator*, to accompany the designation of a prohibitive function.

Since a function, once specified as prohibitive, will be skipped by the verifier until its inputs are inferred concretely, the verifier cannot gather constraints relating the input and output buffers of that function until the inputs can be inferred via other constraints. There are cases, however, where introducing constraints relating the input and output buffers once some *other* subset of them (e.g., the output buffers) are inferred concretely would be useful or, indeed, is central to eventually inferring other prohibitive functions' inputs concretely.

Perhaps the most straightforward example arises in symmetric encryption modes that require the inversion of a block cipher in order to decrypt a ciphertext (e.g., CBC mode). Upon reaching the client SEND instruction for a message, the verifier reconciles the observed client-to-server message msg_n with the constraints σ.cons accumulated on the path to that SEND; for example, suppose this makes concrete the buffers corresponding to outputs of the encryption routine. However, because the block cipher was prohibitive and so skipped, constraints relating the input buffers to those output buffers were not recorded, and so the input buffers remain unconstrained by the (now concrete) output buffers. Moreover, a second pass of the client execution will not add additional constraints on those input buffers, meaning they will remain unconstrained after another pass.

We implemented a feature to address this situation by permitting the user to specify a lazy constraint generator along with designating the block cipher as prohibitive. The lazy constraint generator takes as input some subset of a prohibitive function's input and output buffers, and

produces as output a list of constraints on the prohibitive function's other buffers. The generator is "lazy" in that it will be invoked by the verifier only after its inputs are inferred concretely by other means; once invoked, it produces new constraints as a function of those values. In the case of the block cipher, the most natural constraint generator is the inverse function, which takes in the key and a ciphertext and produces the corresponding plaintext to constrain the value of the input buffer.

More precisely, a lazy constraint generator can be defined as a triple $L = (inE, outE, f)$ as follows:

1. *inE*: A set of symbolic expressions corresponding to the "input" of f.
2. *outE*: A set of symbolic expressions corresponding to the "output" of f.
3. f: A pure function relating *inE* and *outE* such that if f could be symbolically executed, then the constraint $f(inE) = outE$ would be generated.

The set of expressions *outE* corresponds to the output of f, not to the output of the lazy constraint generator. The lazy constraint generator L is blocked until the set of expressions *inE* can be inferred to uniquely take on a set of concrete values (e.g., $inE = 42$). At that point, L is "triggered", generating the real constraint $outE = f(inE)$ that can be added to the path condition (e.g., $outE = f(42) = 2187$). Note that f may correspond to a prohibitive function $p(x)$ or it may correspond to its inverse $p^{-1}(x)$. In the latter case, the expressions *inE* represent the "input" to f but represent the "output" of a prohibitive function $p(x)$.

In order to create a lazy constraint generator, a prohibitive function definition (i.e., `DEFINE_MODEL...`) includes an additional call to the special function:

```
DEF_LAZY(uint8 *inE, size_t inE_len,
         uint8 *outE, size_t outE_len,
         const char *f_name)
```

The `f_name` parameter designates to the verifier which function should be triggered if `inE` can be inferred concretely. Note that since there is only one input buffer and one output buffer, some serialization may be required in order to use this interface, but it is straightforward to wrap any trigger function appropriately.

We illustrate lazy constraint functionality using two example clients. First, let `p` and `p_inv` be the following "prohibitive" function and its inverse.

```
unsigned int p(unsigned int x) {
    return 641 * x;
}

unsigned int p_inv(unsigned int x) {
    return 6700417 * x;
}
```

The correctness of the inversion for $p(x)$ can be seen from the fact that the fifth Fermat number, $F_5 = 2^{32}+1$, is composite with factorization $641 \cdot 6700417$. Since addition and multiplication of 32-bit unsigned integer values is equivalent to arithmetic modulo 2^{32}, we have:

$$\begin{aligned} p^{-1}(p(x)) &= 6700417 \cdot 641 \cdot x \\ &= \left(2^{32}+1\right) \cdot x \\ &\equiv (1) \cdot x \pmod{2^{32}}. \end{aligned}$$

In place of $p(x)$, one could use any function with a well-defined inverse, such as a CBC-mode block cipher.

Fig. 10 shows an example program where p is a prohibitive function. Assume that the lazy constraint generator represents p^{-1}, the inverse function. Suppose that when execution reaches `SEND(y)`, the corresponding data observed over the network is 6410. This implies a unique, concrete value for y, so the lazy constraint generator is triggered, generating a new constraint, $x = p^{-1}(6410) = 10$. Any observed value for `SEND(x)` other than 10 causes a contradiction.

Fig. 11 shows a slightly trickier test case. Here, assume that the lazy constraint generator is defined to correspond to the original function $p(x)$, instead of the inverse. Along the particular branch containing `SEND(y)`, the concretization of variable x is implied by the path condition `(x == 10)` rather than by reconciling symbolic expressions at a network SEND point. The implied value concretization (IVC) of x triggers the execution of $p(x) = p(10) = 6410$ and results in the new constraint $y = 6410$. This constraint will then be matched against the network traffic corresponding to `SEND(y)`. Note that at the time of writing, KLEE did not provide full IVC. As a workaround, we inserted extra solver invocations at each SEND to determine whether the conjunction of the path condition and network traffic yielded enough information to trigger any lazy constraint generators that have accumulated.

```
int main() {
    unsigned int x, y;
    MAKE_SYMBOLIC(&x);
    y = p(x);
    SEND(y);
    SEND(x);
    return 0;
}

// Positive test case: 6410, 10
// Negative test case: 6410, 11
```

Figure 10: Client for which a lazy constraint generator could be triggered due to reconciliation with network traffic at a SEND point.

```
int main() {
    unsigned int x, y;
    MAKE_SYMBOLIC(&x);
    y = p(x);
    SEND(314);
    if (x == 10) {
        SEND(y);
    } else {
        SEND(159);
    }
    return 0;
}

// Positive test case: 314, 159
// Positive test case: 314, 6410
// Negative test case: 314, 6411
```

Figure 11: Client for which a lazy constraint generator could be triggered due to the implied value at a symbolic branch.

Note that our OpenSSL case study in Sec. 5.4 does not require lazy constraint functionality since in the AES-GCM encryption mode, the ciphertext and plaintext buffers are related by simple exclusive-or against outputs from the (still prohibitive) block cipher applied to values that can be inferred concretely from the message. So, once the inputs to the block cipher are inferred by the verifier, the block cipher outputs can be produced concretely, and the plaintext then inferred from the concrete ciphertexts by exclusive-or.

Although the CBC-mode cipher suites in TLS are no longer preferred due to padding-oracle attacks [33], a number of legacy clients still run them, as they lack support for the newer Authenticated Encryption with Associated Data (AEAD) modes such as AES-GCM. Lazy constraint generation enables behavioral verification of these legacy clients.

In the limit, lazy constraint generators could be configured for every single prohibitive function, such that triggering one could cascade into triggering others. This would essentially mimic the functionality of the multi-pass algorithm of Sec. 5. An advantage of this approach would be that the "subsequent passes" (as emulated by cascading lazy constraints) execute only the code that was skipped via the prohibitive function mechanism—other unrelated code does not require re-execution. The tradeoff is an increase in the number of solver queries used to (1) detect that lazy constraints can be triggered, and to (2) stitch the new constraints into the path condition and check whether satisfiability is maintained. Future work could compare the performance tradeoffs between these two mechanisms.

FlexCore: Massively Parallel and Flexible Processing for Large MIMO Access Points

Christopher Husmann[1], Georgios Georgis[1], Konstantinos Nikitopoulos[1], and Kyle Jamieson[2]

[1]5G Innovation Centre, Institute for Communication Systems, University of Surrey
[2]Princeton University and University College London

Abstract

Large MIMO base stations remain among wireless network designers' best tools for increasing wireless throughput while serving many clients, but current system designs, sacrifice throughput with simple linear MIMO detection algorithms. Higher-performance detection techniques are known, but remain off the table because these systems parallelize their computation at the level of a whole OFDM subcarrier, sufficing only for the less-demanding linear detection approaches they opt for. This paper presents FlexCore, the first computational architecture capable of parallelizing the detection of large numbers of mutually-interfering information streams at a granularity below individual OFDM subcarriers, in a nearly-embarrassingly parallel manner while utilizing any number of available processing elements. For 12 clients sending 64-QAM symbols to a 12-antenna base station, our WARP testbed evaluation shows similar network throughput to the state-of-the-art while using an order of magnitude fewer processing elements. For the same scenario, our combined WARP-GPU testbed evaluation demonstrates a 19× computational speedup, with 97% increased energy efficiency when compared with the state of the art. Finally, for the same scenario, an FPGA-based comparison between FlexCore and the state of the art shows that FlexCore can achieve up to 96% better energy efficiency, and can offer up to 32× the processing throughput.

1 Introduction

One of the most important challenges in the design of next-generation wireless communication systems is to meet users' ever-increasing demand for capacity and throughput. Large Multiple Input-Multiple Output (MIMO) systems with spatial multiplexing are one of the most promising ways to satisfy this demand, and so feature in emerging cellular [1] and local-area [21, 22] networking standards. For example, in LTE-Advanced and 802.11ac, up to eight antennas are supported at the access point (AP).

In a system employing spatial multiplexing, multiple transmit antennas send parallel information streams concurrently to multiple receive antennas. While the technique can be used both in the uplink and the downlink of multi-user MIMO networks, here we focus on the uplink case, where several users concurrently transmit to a multi-antenna AP. The need for high throughput in the uplink stems from the emergence of new applications for wireless networks, such as video internet telephony, wireless data backup, and Internet of Things devices, that have shifted the ratio between uplink and downlink traffic quantities towards the former.

However, simply having enough antennas at the AP does not always suffice: to fully realize MIMO's potential throughput gains, the AP must effectively and efficiently demultiplex mutually-interfering information streams as they arrive. Current large MIMO AP designs such as Argos [38], BigStation [54] and SAM [41], however, use linear methods such as *zero-forcing* and *minimum mean squared error* (MMSE). These methods have the advantage of very low computational complexity, but suffer when the MIMO channel is poorly-conditioned, as is often the case when the number of user antennas approaches the number of antennas at the AP [32].

On the other hand, *Sphere Decoder*-based MIMO detectors can boost throughput over linear methods significantly [8, 32], even in cases where the number of user antennas approaches the number of AP antennas. The cost of such methods, however, is their increased computational complexity: compute requirements increase exponentially with the number of antennas [19, 24], soon becoming prohibitive. Indeed, processing complexity is a significant issue for any advanced wireless communication system. While the clock speed of traditional processors is plateauing [12], emerging hardware architectures including GPUs support hundreds of cores, presenting an opportunity to parallelize the processing load, along with the challenge of how to do so most efficiently. BigStation [54] is an example of a system that handles the

processing load of a large MIMO system using multiple commodity servers, exploiting parallelism down to the OFDM subcarrier level. Since BigStation uses zero-forcing methods to decode clients' transmissions, this level of parallelism suffices for the AP to be able to decode multiple incoming transmissions and send acknowledgement frames back to the client without letting the wireless medium become idle for more than the amount of time prescribed by the 802.11 standards [22]. The cost, however, is diminished wireless performance, due to linear zero-forcing detection.

Antennas	Throughput	Complexity
$\mathbf{2 \times 2}$	45 Mbit/s	1.2 GFLOPS
$\mathbf{4 \times 4}$	100	13
$\mathbf{6 \times 6}$	162	105
$\mathbf{8 \times 8}$	223	837

Table 1— A summary comparison of the throughput achieved and computational rate required for a Sphere Decoder implementation [32].

Table 1 shows the number of floating point operations per second that a single processing core must maintain to perform optimal, *maximum-likelihood*, depth-first Sphere decoding,[1] as in [32], when processing OFDM symbols at the same rate they arrive over the air, for typical Wi-Fi system parameters.[2] The table is parametrized on the size of the MIMO system (number of clients times number of AP antennas), highlighting the exponential increase in floating point operations per second required to keep up with linearly-increasing numbers of clients. By the time a Sphere decoder reaches eight clients, it must meet a rate on the order of 10^3 GFLOPS, saturating, for example, Intel's Skylake core i7 architecture, whose arithmetic subsystem achieves an order of magnitude less computational throughput [23]. Clearly, the answer to this performance mismatch is to decompose and parallelize the problem, and the aforementioned previous work has made progress in this direction, dedicating one core to each OFDM subcarrier. But as the number of clients grows, because of the mismatch between the exponential growth in required computational rate and the much slower growth of the frequency bandwidth Wi-Fi systems use in general, there is a need for a new family of MIMO detectors that allows parallelization *below* the subcarrier level using a small number of parallel processing elements.

To meet the processing needs of large, high-performance MIMO APs, we present *FlexCore*, an asymptotically-optimal, massively-parallel detector for large MIMO systems. FlexCore reclaims the wasted throughput of linear detection approaches, while at the same time parallelizing processing below the subcarrier level, thus enabling it to meet the tight latency requirements required for packetized transmissions. In contrast to existing low-complexity Sphere decoder architectures [4], FlexCore can exploit any number of available processing elements and, for however many are available, maximize throughput by allocating processing elements only to the parts of the decoding process most likely to increase wireless throughput. Consequently, as additional processing elements are provisioned, FlexCore continues to improve throughput. By this design, FlexCore can avoid unnecessary computation, allocating only as many processing elements as required to approach optimal (*maximum-likelihood*) MIMO wireless throughput performance. FlexCore's computation proceeds in a nearly "embarrassingly" parallel manner that makes parallelization efficient for a broad set of implementation architectures, in particular GPUs, even allowing parallelization across devices. To achieve this, we present novel algorithms to:

1. Choose which parts of the Sphere decoder tree to explore, through a "pre-processing" step, and then,
2. efficiently allocate the chosen parts of the Sphere decoder tree to the available processing elements.

FlexCore's *pre-processing* step is a low-overhead procedure that takes place only when the transmission channel significantly changes. It narrows down which parts of the Sphere decoder tree the system needs to explore to decode the clients' transmissions, *i.e.*, find a *solution* to the decoding problem. The pre-processing step identifies the "most promising" candidate solutions in a probabilistic manner, and occurs *a priori*, without knowing the signals received from the clients themselves, but instead based on the knowledge of the transmission channel and the amount of background noise present. In this part of the design, FlexCore introduces a new probabilistic model to identify the most promising candidate solutions and an indexing technique wherein the tree nodes are labeled by *position vectors*. We also introduce a novel pre-processing tree structure and tree search distinct from the traditional Sphere decoder tree search. These new techniques allow us to efficiently identify the most promising candidate solutions.

FlexCore's *allocation* step maps each of the chosen paths in the Sphere decoder's search tree to a single processing element, spreading the load evenly. While this previously required redundant calculations across parallel tasks as well as multiple sorting operations, FlexCore skips them by introducing a new node selection strategy.

Roadmap. The rest of this paper is structured as follows. We start with a primer of the Sphere decoder in §2, followed by a description of its design in §3. In §4 we present our implementation strategy on both GPUs and FPGAs, highlighting FlexCore's computational flexibility

[1] Simulated results for Rayleigh channel, 16-QAM and 13 dB SNR.

[2] 20 MHz frequency bandwidth, resulting in a number of OFDM subcarriers N_c on the order of 50.

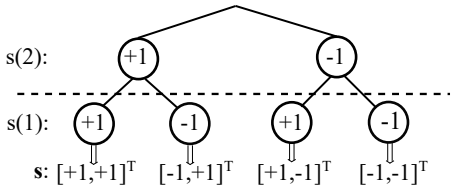


Figure 1— Sphere decoding tree for two transmit antennas, each sending a binary-modulated wireless signal.

across different platforms. Section 5 presents our algorithmic performance evaluation based on both over-the-air experiments and trace-based simulations. In the same section, separate GPU- and FPGA-based evaluations follow, and the applicability of our FlexCore implementations to LTE in terms of computation time is discussed. In §6 we discuss relevant related work, before concluding in §7.

2 Primer: The Sphere Decoder

In order to put FlexCore's high-level design into context, we now introduce the basic principles of the Sphere decoder, then briefly describe the *fixed complexity sphere decoder* (FCSD), an approximate Sphere decoder whose design is amenable to parallelization.

When transmitting a symbol vector $\mathbf{s}$ in an OFDM-MIMO system with N_t transmit and N_r receive antennas (*i.e.*, $N_t \times N_r$ MIMO), with $N_r \geq N_t$, the vector of data arriving at the receive antennas on a particular OFDM subcarrier is $\mathbf{y} = \mathbf{Hs} + \mathbf{n}$, with $\mathbf{H}$ being the $N_r \times N_t$ MIMO channel matrix. The N_t elements of the transmit vector $\mathbf{s}$ belong to a complex constellation Q (*e.g.*, 16-QAM) consisting of $|Q|$ elements. The vector $\mathbf{n}$ represents an N_r-dimensional white Gaussian noise vector. The Sphere decoder transforms the *maximum-likelihood* (ML) problem[3] into an equivalent tree search [44]. In particular, by a QR-decomposition of the MIMO channel matrix ($\mathbf{H} = \mathbf{QR}$, where $\mathbf{Q}$ is an orthonormal and $\mathbf{R}$ an upper triangular matrix), the ML problem can be transformed into $\hat{\mathbf{s}} = \arg\min_{\mathbf{s}\in|Q|^{N_t}} \|\bar{\mathbf{y}} - \mathbf{Rs}\|^2$, with $\bar{\mathbf{y}} = \mathbf{Q}^*\mathbf{y}$. The corresponding tree has a height of N_t and a branching factor of $|Q|$. As shown in Fig. 1 for a 2×2 MIMO system with binary modulation, each level l of the tree corresponds to the symbol transmitted from a certain antenna. Each node in a certain level l is associated with a partial symbol vector $\mathbf{s}_l = [s(N_t - l), \ldots, s(N_t)]^T$ containing all possibly-transmitted symbols from senders down to and including this level, and is characterized by its partial Euclidean distance [44]

$$c(\mathbf{s}_l) = \left[y(l) - \sum_{p=l}^{N_t} R(k,p) \cdot s(p) \right]^2 + c(\mathbf{s}_{l+1}), \qquad (1)$$

with $R(k,p)$ being the element of $\mathbf{R}$ at the k^{th} column and the p^{th} row, and $c(\mathbf{s}_{N_t+1}) = 0$. The ML problem then transforms into finding the tree leaf node with the minimum $c(\mathbf{s}_1)$; the corresponding tree path $\mathbf{s}_1$ is the ML estimate. Many approaches explore the Sphere decoding tree in a depth-first order via which, in contrast to breadth-first approaches, they can guarantee ML performance, and they can adjust their complexity to the channel condition [8, 14, 32]. However, depth-first tree search is strictly sequential, making parallelization extremely difficult: a naive parallel evaluation of all Euclidean distances in order to minimize processing latency would be impractical. For example, for an 8×8 MIMO with 64-QAM modulation, this approach would require performing 2.8×10^{14} Euclidean distance calculations in parallel.

Fixed Complexity Sphere Decoder. The *Fixed Complexity Sphere Decoder* (FCSD) [4] is an approximate Sphere decoder algorithm: instead of examining (and pruning) all possible Sphere decoder leaf nodes (*i.e.*, all possible solutions), the FCSD visits only a predefined set of them. Specifically, the FCSD visits all nodes at the top L levels of the tree, while for the remaining $N_t - L$ levels, only visits the child node with the smallest partial Euclidean distance (*i.e.*, the branching factor is reduced to one). Since the FCSD visits a predefined set of leaf nodes, it can visit all these in parallel, returning the leaf node with the minimum partial distance as its final answer.

There are however, three important drawbacks to the FCSD's approach. First, the required number of parallel processes has to be a power of the order of the QAM constellation, so the FCSD cannot efficiently adjust to the number of the available processing elements. Second, the visited tree paths are not necessarily the ones that are the most likely to constitute the correct solution, which means that much of the available processing power is not efficiently allocated. Finally, the FCSD cannot differentiate between favorable channel conditions, where even linear approaches would give near-ML performance, and unfavorable channel conditions, where linear approaches are not efficient. In the next section, we describe FlexCore's techniques to address this problem by visiting just the most promising tree paths, maximizing throughput for a given number of processing elements and exploiting any number of available processing elements, not just numbers that are a power of the constellation size.

3 Design

As shown in Fig. 2, FlexCore's architecture consists of two major components: the *pre-processing* module which identifies the most promising tree paths as a function of the MIMO channel and the background noise power, and the *parallel detection* module that actually allocates tree paths to processing elements when the AP decodes an incoming signal. Since FlexCore evaluates the most promising paths by accounting for the transmission channel and background noise, it needs to re-execute pre-processing

[3] In other words, the problem of finding the most likely set of symbols the transmitting antennas sent.

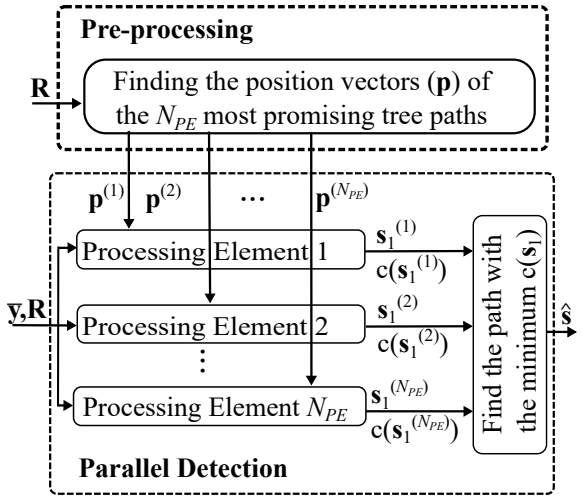


Figure 2— Block Diagram of FlexCore.

only when the transmission channel changes, similarly to the QR decomposition that is required for Sphere decoding. However, as we show below, the delay introduced by pre-processing is insignificant compared to that of the QR decomposition.

3.1 Pre-processing module

FlexCore's pre-processing uses the notion of a *position vector* $\mathbf{p}$, which uniquely describes all the possible tree paths relative to the (unknown) received signal. The position vector is of equal size to the Sphere decoder tree height: each of its elements $p(l)$ takes an integer value ranging from 1 to $|Q|$ that describes the position of the corresponding node at the lth level of the Sphere decoder tree as a function of its index, when sorting the nodes in ascending Euclidean distance order.

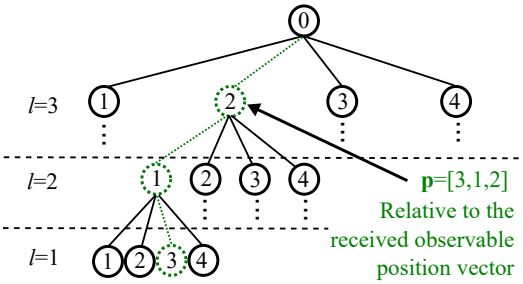


Figure 3— Sorted search for tree for 3 transmit antennas and 4-QAM modulation. The path with the the position vector $\mathbf{p} = [3,1,2]$ is highlighted (green, dashed).

Independent channel example. To illustrate the basic principle behind pre-processing, we present the following simplified example. Suppose that a vector $\mathbf{s}$ of two binary symbols is transmitted via two independent Gaussian noise channels, with noise powers σ_l^2 with $l = 1,2$ and $\sigma_2^2 \geq \sigma_1^2$, with the symbol $s(l)$ being transmitted

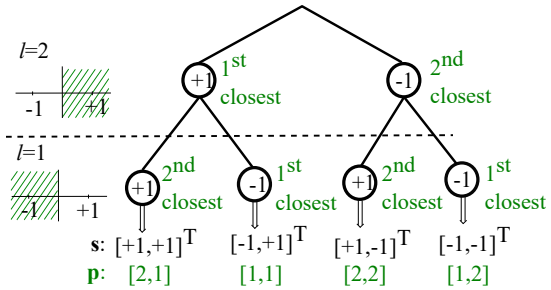


Figure 4— FlexCore's pre-processing and most-promising path selection for the **independent channel example** (two transmit antennas, binary modulation).

through the channel l. In FlexCore each parallel channel corresponds to a level of the Sphere decoding tree.[4] All possible transmitted symbol combinations are shown in Fig. 4. It is known from detection theory that the best decoding method for this example is to choose the symbols lying on the same side of the x-axis (positive or negative) as the received signals lie. That means that the most likely solution in the tree of Fig. 4 is the path consisting of the first-closest symbols to the received signal (on each parallel channel) and, therefore, its position vector is $\mathbf{p} = [1,1]$. It can also be shown that the corresponding probability of including the correct vector is $P_c(\mathbf{p} = [1,1]) = (1 - P_e(1))(1 - P_e(2))$, with $P_e(l)$ being the error probability of binary modulation for a noise variance of σ_l^2. The second most likely path to include the correct solution is for a case where on one parallel channel the symbol lies on the same side of the x-axis as the received observable (*i.e.*, the tree path includes the closest symbol to the received observable) and on the other channel the symbol and received observable lie on different sides of the x-axis (*i.e.*, the tree path includes the second-closest symbol to the received observable). This means that the second-most-promising path is either the path $\mathbf{p} = [1,2]$ or the path $\mathbf{p} = [2,1]$.[5] Finally, the least-promising path is $\mathbf{p} = [2,2]$ with $P_c(\mathbf{p} = [2,2]) = P_e(1) \cdot P_e(2)$.[6]

After performing the pre-processing step, and when the actual received signal $\mathbf{y}$ is available, detection takes place. In the case that our system has only two available processing elements, FlexCore calculates the Euclidean distances for the two most promising paths $\mathbf{p} = [1,1]$ and $\mathbf{p} = [1,2]$. Then, the detection output is the vector with the smallest calculated Euclidean distances (Fig. 2).

Generalizing to the MIMO channel. In a similar manner to the independent Gaussian channels case, for the actual Sphere decoding tree, and for any QAM constella-

[4] In the Sphere decoding case, however, the levels are not independent but, as we show in the Appendix, the following approach still applies.

[5] Since we assumed that $\sigma_2^2 \geq \sigma_1^2$, then $P_c(\mathbf{p} = [1,2]) \geq P_c(\mathbf{p} = [2,1])$ and therefore $(1 - P_e(1))P_e(2) \geq P_e(1)(1 - P_e(2))$.

[6] We note again that the position vector identifies these tree paths in terms *relative* to the signal that the AP will later receive, instead of identifying absolute tree paths, hence pre-processing is possible *a priori*.

tion, the corresponding probabilities can be approximated as

$$P_c(\mathbf{p}) \approx \prod_{l=1}^{N_t} P_l\left(p(l)\right) \tag{2}$$

with

$$P_l\left(p(l)\right) = \left(1 - P_e(l)\right) \cdot \left(P_e(l)\right)^{(p(l)-1)} \tag{3}$$

and

$$P_e(l) = \left(2 + \frac{2}{\sqrt{|Q|}}\right) \cdot \mathrm{erfc}\left(\frac{|R(l,l)| \cdot \sqrt{E_s}}{\sigma}\right) \tag{4}$$

where *erfc* is the complementary error function, which can be calculated on-the-fly or pre-calculated using a look-up table, E_s is the power of the transmitted symbols, σ^2 is the noise variance, and $p(l)$ is the l^{th} element of $\mathbf{p}$. We defer a mathematical justification to the Appendix.

3.1.1 Finding the most promising position vectors

FlexCore needs to identify the set $\mathscr{E}$ consisting of the N_{PE} most promising position vectors, with N_{PE} being the number of available processing elements. An exhaustive search over all possible paths becomes intractable for dense constellations and large antenna numbers. Therefore, we translate the search into a new *pre-processing tree* structure (distinct from the Sphere decoder tree) and propose an efficient traversal and pruning approach that substantially reduces pre-processing complexity.

We now explain how to construct and traverse the pre-processing tree with an example: Fig. 5 shows its structure for three transmit antennas. Each node in the tree can be described by a position vector and its likelihood P_c (Eq. 2). Tree construction begins by setting the tree

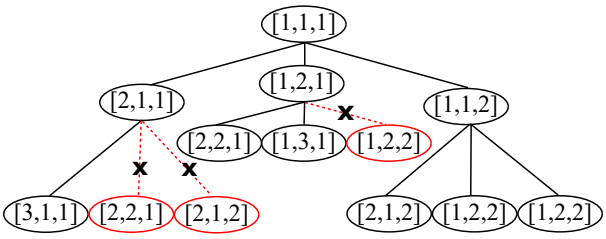


Figure 5— A *pre-processing tree* construction for three transmit antennas.

root to a node whose position vector consists of only ones ($\mathbf{p} = [1,1,\ldots,1]$) since this will always be the "most promising" one, regardless of the MIMO channel. Then we expand the tree root node, namely, we construct its child nodes and calculate their P_c values. To find the w^{th} child node ($w \in [1,\ldots,N_t]$), we increment the w^{th} element of the parent's position vector by one, as shown in Fig. 5. To avoid duplication of pre-processing tree nodes, when expanding a node whose position vector has been generated by increasing its l^{th} element, the w^{th} children nodes ($w \in [l+1,\ldots,N_t]$) are not expanded. To avoid unnecessary computations while calculating the P_c values of the children nodes, we further observe that for two position vectors $\tilde{\mathbf{p}}$ and $\mathbf{p}$ that only differ in their w^{th} component, their probabilities are related by $P_c(\tilde{\mathbf{p}}) = P_c(\mathbf{p}) \cdot P_e(w)$. After expanding the tree root, we include its position vector in the set $\mathscr{E}$ of most promising position vectors and we store all the children nodes of the expanded node and their P_c values in a sorted candidate list $\mathbb{L}$ (of descending order in P_c). Tree traversal continues by expanding the node with the highest P_c value in $\mathbb{L}$. The expanded node is then removed from $\mathbb{L}$, its position vector is added to $\mathscr{E}$ and the node's children are appended to $\mathbb{L}$. Whenever $|\mathbb{L}|$ exceeds N_{PE}, we remove from $\mathbb{L}$ the $|\mathbb{L}| - N_{PE}$ nodes with the lowest P_c values. The process continues until $|\mathscr{E}| = N_{PE}$. Finally, we introduce a *stopping criterion* in order to terminate the tree search if the sum of the P_c values of the vectors currently in $\mathscr{E}$, is larger than a predefined threshold. An example case is discussed in Section 5.

Pre-processing complexity. In terms of this process' complexity, we first calculate the error probabilities ($P_e(l)$ in Eq. 4), which can be computed once and reused several times during pre-processing. Then, we require in the worst case N_t real multiplications per expanded node. Since the maximum amount of expanded nodes is at most N_{PE}, the maximum complexity in terms of real multiplications is $(N_{PE} \cdot N_t)$. In very dense constellations (*e.g.*, 256-, 1024-QAM) a rather large number of parallel processing elements may be required to reach near ML-performance. In such challenging scenarios, sequential execution of the pre-processing phase may introduce a significant delay. However, our simulations have shown that a parallel expansion of the nodes with the highest probabilities P_c in $\mathbb{L}$ is possible with negligible throughput loss compared to a sequential implementation, provided that the ratio of available processing elements N_{PE} to the number of nodes expanded in parallel is greater than ten. As a result, the latency of the pre-processing step for large MIMO systems is insignificant compared to that of the QR decomposition. In MIMO systems with dynamic channels and user mobility, the most promising paths will vary in time. Therefore, and as validated in [17], in such cases reliable channel estimates are still required to preserve the gains of spatial multiplexing. FlexCore will then leverage these estimates to recalculate the most promising paths, together with the traditionally required channel-based pre-processing (*e.g.*,

	Pre-Processing			Detection	
	QR/ZF	FlexCore		FlexCore	
		$N_{PE}=32$	$N_{PE}=128$	$N_{PE}=32$	$N_{PE}=128$
8×8	≈2048	102	301	4608	18432
12×12	≈6912	136	391	9984	39936
Parallelizability	-	3	12	32	128

Table 2— Complexity in real multiplications and "parallelizability" of Pre-Processing and FlexCore detection.

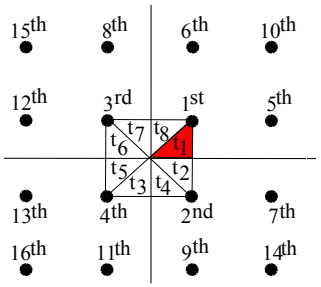


Figure 6— Detection square and triangles 1-8 for 16-QAM and approximate predefined symbol ordering, calculated when the received symbol is within triangle t_1.

channel inversion for linear detection or QR decomposition for Sphere decoder-based detection). In such dynamic channels, pre-processing complexity requirements can become comparable to those of FlexCore's detection, as shown in Table 2. Latency requirements can then be determined by the required sequential QR decomposition (or channel inversion for linear detection).

3.2 Core Allocation and Parallel Detection

To perform detection in parallel, each of the calculated position vectors in $\mathscr{E}$ has to be allocated to a different processing element. FlexCore's pre-processing has already identified the position vectors as a function of their Euclidean distance sorted order. For example, if the corresponding position vector is $\mathbf{p} = [3,1,2]$, then the tree path to be processed consists of the node with the second smallest Euclidean distance at the top level, the smallest in the second level and the third smallest in the first level. Typically, finding the node (*i.e.*, QAM symbol) with the third smallest Euclidean distance to the observable would require exhaustive calculation of all Euclidean distances at a specific level (*e.g.*, for 64-QAM it would require 63 unnecessary Euclidean distance calculations). In order to avoid these unnecessary computations, we exploit the symmetry of the QAM constellation, defining an approximate predefined order based on the relative position of the "effective" received point $\tilde{y}_l$ in the QAM constellation.

$$\tilde{y}_l = \left(y_l - \sum_{p=1+l}^{N_t} R(l,p) \cdot s(p) \right) / R(l,l). \qquad (5)$$

We calculate the approximate predefined symbol order by assuming that the "effective" received point lies in a square which is centered at the center of the QAM constellation and of side length equal to the minimum distance between consecutive constellation symbols as in Fig. 6. We then split the square into eight triangles t_i ($i = 1,...,8$) and via computer simulations, compute the most frequent sorted order for "effective" received points lying in these triangles (as a function of their relative position to the center of the square), storing the order in a look-up table. Fig. 6 shows the resulting approximate order for a 16-QAM constellation when the received point lies within t_1. We note that the constellation's symmetrical properties allow us to store the order of just a single triangle (*e.g.*, for t_1) since the order for all other triangles will be just circularly shifted (with a center one of the constellation points). During actual decoding, at each level we identify the relative position of the square as well as the relative position of the received point within the square. We then identify the symbol with the k^{th} smallest distance by using the predefined ordering. If, however, the latter points to a symbol that is not part of the constellation (*i.e.*, the center of the grid is not the same as the center of the constellation), then the corresponding Euclidean distance calculation unit is deactivated.

4 Implementation

To showcase the versatility and efficiency of FlexCore we implement it on FPGAs and GPUs. In particular, since FlexCore focuses available processing resources to the most promising parts of the Sphere decoding tree, we demonstrate that it can consistently outperform state-of-the-art implementations, regardless of the underlying platform. We first evaluate FlexCore's gains when realized on GPUs based on the Compute Unified Device Architecture (CUDA) programming model [31]. These types of many-core architectures are among the more challenging technologies to display FlexCore's actual gains, as the developer does not have direct control over the allocation of processing elements, but can instead control the way parallel threads are generated.

Additionally, we implement FlexCore on FPGAs, a platform that is less programmable than a GPU but more able to be tailored to FlexCore's design. FPGA implementation of fixed-complexity detection schemes has been examined in the literature [2, 5, 26, 49], enabling low latency and high throughput detection. Apart from high performance at a low power envelope [16], FPGAs allow greater flexibility for the examined fixed complexity schemes. For instance, the designer may choose to implement in parallel a fraction or even a single path of the total paths required. Therefore, FPGAs are more effective in highlighting FlexCore's computational flexibility.

GPU-based detector implementation. To implement FlexCore we have extended the MIMOPACK library [34] by introducing support for single-precision floating-point computations, and therefore we have achieved improved processing and memory transfer throughput, and reduced storage requirements. To further improve transfer throughput we added support for non-pageable host memory allocation. Finally, we provided support for streams, a means of concurrent execution on GPUs through overlapping asynchronous (i.e., in practice independent) operations.

The parallel FCSD generates $N_{sc} \times |Q|^L$ threads on the GPU, with L being the number of levels to be fully ex-

panded and N_{sc} the number of subcarriers to be processed in parallel, given it is supported by the memory of the GPU. To facilitate parallel computations, storage for all position vectors is allocated in advance by the library. Our FlexCore implementation generates $N_{sc} \times |\mathcal{O}|$ threads. We note that FlexCore has a higher workload compared with FCSD due to the additional arithmetic/branching operations and their application to the topmost level of the tree. Memory-wise, compared to MIMOPACK's FCSD our FlexCore implementation requires three additional Host to Device (H2D) transfers: a) two $|Q|\times$4-byte wide involving the order of a single triangle and b) one $N_{sc} \times N_t \times |\mathcal{O}|$-byte wide containing the tree paths. Since the latter matrix essentially represents the position vectors, its values can be limited to single bytes for $|Q| \leq 256$.

FPGA-Based Detection Design. FlexCore was designed with latency minimization in mind and thus, we present a pipelined parallel architecture and the implementation of both FCSD's and FlexCore's detection engines. For the purpose of consistency, we consider our processing element as the fully-instantiated logic required to process a whole Sphere decoder path from the top to the bottom level of the tree. To implement the fixed-complexity schemes, we have designed modular, low-complexity and highly parameterizable fixed-point architectures using Verilog RTL code. We have explicitly designed each branch at every level, replicated branches and connected levels in a structural manner. In order to save resources, in the topmost FCSD level we employ constant complex coefficient multipliers (CCMs) which perform a hard-wired integer multiplication of $R(l,l)$ by the complex constellation point value. Moreover, to avoid the division in Eq. 5, we consider the effective received point as $\tilde{y}_l \cdot R(l,l)$, (for all comparisons involving the constellation plane, we multiply the latter's values by $R(l,l)$). To save DSP48 resources (Xilinx's embedded FPGA multiplication/arithmetic logic unit) and to maintain a low utilization of the generic FPGA fabric, we employ multiple constant coefficient multipliers (MCMs) similar to the ones described in [11] (*i.e.*, employing indices for **s**).

Fig. 7 shows an overview of the detection engines' architecture and modular design principle which allows for reusability, fair comparison and an instantiation of an arbitrary number of detection paths. Bit widths and pipeline levels are parameterizable at instantiation time. All levels of FlexCore are implemented by replicating the branches displayed in the top left of Fig. 7. The FCSD's design follows a very similar approach allowing the reuse of FlexCore's modules, apart from the topmost FCSD level, where for its fully parallel implementation we designed and instantiated $|Q|$ CCMs in parallel. Note that the high level architecture of every branch remains almost identical to the one displayed in Fig. 7, apart from its $\tilde{y}_l \cdot R(l,l)$ unit, whose complexity increases as we approach the bottom tree levels. Additionally to the FCSD, FlexCore's branches include pipeline registers to store the desired closest point offset to the received vector and non-pipelined registers storing the order for a single triangle. FlexCore's slicer computes the midpoint value and index instead of the actual constellation point, forwarding the results to the FlexCore engine which outputs the detected constellation point index. The l^2 norm unit (Fig. 7, bottom-left) is designed to directly employ two cascaded DSP48 FPGA slices, one in multiplication mode and the second in multiply-add mode. Finally, a parameterizable (regarding both the supported width and the number of elements) pipelined minimum tree unit outputs the detected solution. At a minimum level of pipeline (*i.e.*, submodule i/o and three registers per embedded multiplier), the total FCSD latency is 95 up to 150 clock cycles ($N_t = 8$ and 12 respectively). The FlexCore engine induces an additional minimum latency of 5 cycles per level.

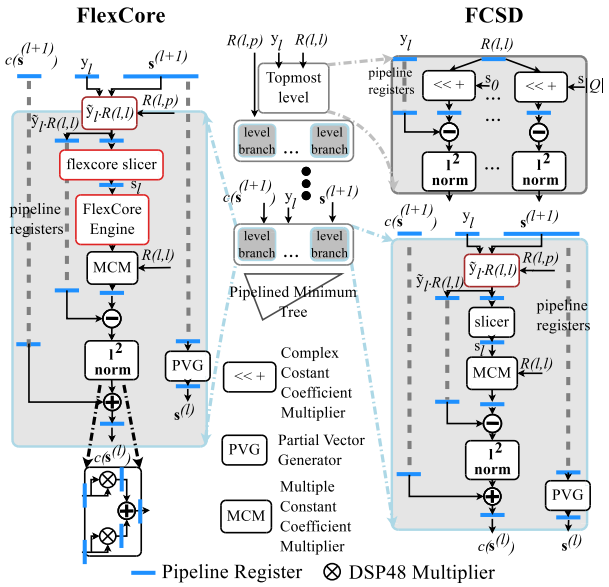


Figure 7— FPGA architecture of proposed detection engines: FlexCore (left) and FCSD (right).

5 Evaluation

In this section we discuss FlexCore's algorithmic performance and implementation aspects compared to the state-of-the-art. We first evaluate FlexCore's throughput performance on our WARP v3 testbed. Based on these results, we then jointly assess FlexCore's algorithmic and GPU implementation performance. Finally, we provide a design space exploration of high-performing detection for various parallelization factors on the state-of-the-art Xilinx Virtex Ultrascale xcvu440-flga2892-3-e FPGA. Our evaluation focuses on scenarios where the channel is static over a packet transmission and it does not account for the pre-processing complexity. Since the pre-processing task needs to take place any time the channel changes (see

§3), the corresponding overhead can be easily calculated based on the assumed channel dynamics.

5.1 Throughput Evaluation

Methodology and setup. To evaluate FlexCore's throughput gains we use Rice's WARP v3 radio hardware and WARPLab software. We employ 16- and 64-QAM modulation with the 1/2 rate convolutional coding of the 802.11 standard. Each user transmits 500-kByte packets over 20 MHz bandwidth channels within the 5 GHz ISM band in indoor (office) conditions. We implement an OFDM system with 64 subcarriers, 48 of which are used to transmit payload symbols, similarly to the 802.11 standard. Eight- and 12-antenna APs are considered, with the distance between co-located AP antennas to be approximately 6 cm. For the eight-antenna AP case, evaluations have been made purely by over-the-air experiments involving all necessary estimation and synchronisation steps (*e.g.*, channel estimation). For the 12-antenna AP case, and due to restrictions on the available hardware equipment, evaluation is performed via trace-driven simulation. To collect the corresponding MIMO channel traces, we have separately measured (over the air) and combined the received channel traces of single-antenna users to 12-antenna APs (1×12) . Fig. 8 displays a graphical overview of our testbed with the positions of the eight- and 12- antenna APs. Similarly to [32], the individual SNRs of the scheduled users differ by no more than 3 dB. This minimizes the condition number of the channel (a low condition number is an indicator of a favorable channel) but therefore also limits the potential gains of FlexCore and Sphere decoding approaches in general. For all our evaluations, the examined SNR is such that an ML decoder reaches approximately the practical packet error rates (PER_{ML}) of 0.1 and 0.01 when the number of active users is equal to the number of the AP antennas. For the realization of both FlexCore and FCSD we employ both the sorted QR decompositions of [4] and [13] and we show the best achievable throughput.

FlexCore's throughput for $N_t = N_r$. Fig. 9 shows the achievable network throughput of FlexCore, FCSD and the trellis-based parallel decoder introduced in [50], for several numbers of available processing elements, against the throughput achieved by exact ML detection and linear MMSE detection. The evaluation is based on the assumption that minimum latency is targeted. Therefore, each parallel element is only allocated to one parallel task. In the case of FlexCore and FCSD, each processing element is used to calculate the Euclidean distance of a single tree path per received MIMO vector. In [50] each processing element calculates the partial Euclidean distance of each constellation point. As a result, [50] would also require a fixed number of processing elements, equal to the QAM constellation's size. In practice, for all schemes, a processing element could be used multiple times to carry out multiple parallel tasks sequentially, but this would result in an increase in latency. In agreement with the literature [32, 38, 54], Fig. 9 demonstrates that linear detection results in a poor throughput when $N_t = N_r$.[7] It also shows that while the trellis-based method of [50] outperforms MMSE, it is consistently inferior to FCSD and FlexCore, in all evaluated scenarios. In addition, it requires a fixed number of processing elements, and is therefore unable to scale its performance with the number of available processing elements. Due to these limitations, in the rest of the paper we focus on comparing FCSD and FlexCore.

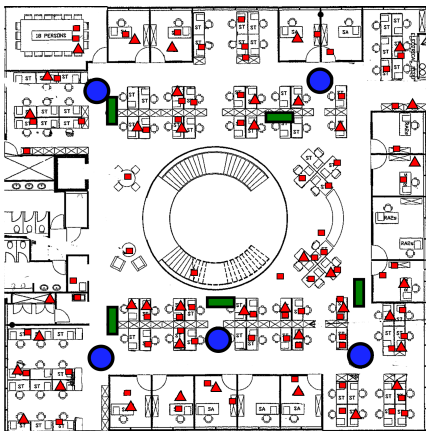


Figure 8— Testbed floorplan (circles: 8-antenna APs, rectangles: 12-antenna APs, triangles: single-antenna users transmitting to 8-antenna APs, squares: single-antenna users transmitting to 12-antenna APs).

Fig. 9 shows that, in contrast to FSCD, FlexCore operates for any number of available processing elements and it consistently improves throughput when increasing the available processing elements. On the other hand, and as discussed in §2, the FCSD can fully exploit processing elements as long as their number is a power of the order of the employed QAM constellation. Figure 9 also shows that for a given number of available processing elements, FlexCore consistently outperforms FCSD in terms of throughput. When 12 users transmit 16-QAM symbols to a 12-antenna AP, at an SNR such that $PER_{ML} = 0.1$ (SNR=13.5 dB), when 196 parallel elements per subcarrier are available, FlexCore can provide nearly $2.5\times$ the throughput of the FCSD. In addition, due to FlexCore's pre-processing, which focuses the available processing power to the tree paths that are most likely to increase wireless throughput, FlexCore requires significantly fewer processing elements than FCSD to reach the same throughput. For example, in a 12×12 64-QAM MIMO system and at an SNR such that $PER_{ML} = 0.01$ (SNR=21.6 dB), FlexCore requires 128 parallel paths to

[7] We note that MMSE can achieve better throughput if we allow $N_t < N_r$. This is shown in [32], as well as in Fig. 10.

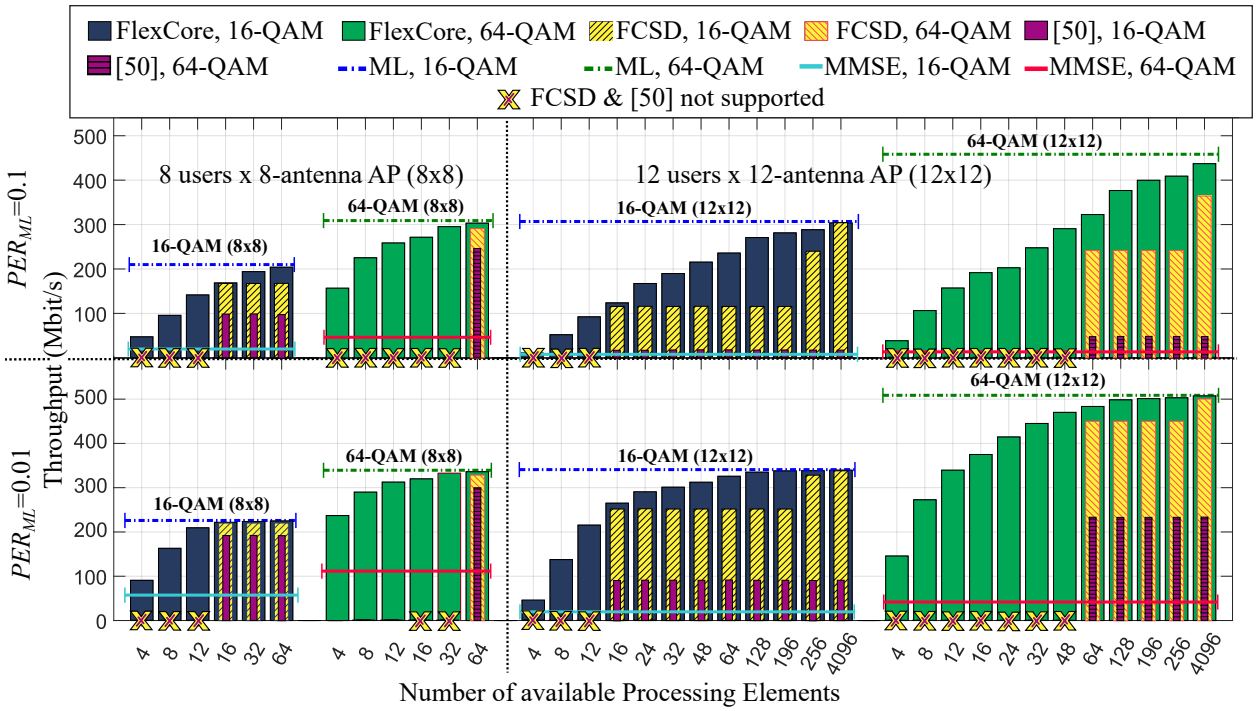


Figure 9— Achievable network throughput of FlexCore, FCSD and trellis-based decoder [50] for minimum processing latency, as a function of the available processing elements, compared to optimal (ML) detection and MMSE.

reach 95% of the ML-bound, whereas FCSD requires 4096. Fig. 9 also shows that, in principle, the gains of FlexCore against the FCSD increase when the transmission conditions become more challenging. Namely, when the number of antennas and the QAM constellation's order increase, and when the SNR decreases (and therefore the PER of the ML solution increases).

FlexCore's throughput v. number of users. The bars in Fig. 10 show the achieved network throughput of FlexCore against the throughput of Geosphere and MMSE, as a function of the number of active users simultaneously transmitting 64-QAM symbols to a 12-antenna AP at an SNR such that $PER_{ML} = 0.01$ (SNR=21.6 dB). We assume that 64 processing elements per sub-carrier are available for FlexCore. However, we also consider an adjustable version of FlexCore (*a-FlexCore*) that from the 64 available processing elements, uses as many as required so that the sum of the P_c values of the corresponding most promising paths becomes 0.95. As expected [38,54], Fig. 10 shows that MMSE is almost optimal only when the number of active users is significantly smaller than the number of AP antennas. In contrast, exact or approximate ML methods, including FlexCore, can support numbers of users that are similar to the number of the AP antennas and still scale network throughput. This ability to reclaim the unexploited throughput of linear detectors separates FlexCore from prominent large MIMO architectures such as [38,54]. Fig. 10 also shows that in contrast to the previously proposed parallel schemes, FlexCore has the ability to adjust the number of activated processing elements and therefore the overall complexity to the channel conditions. When the number of active users is significantly smaller than the one of the AP antennas, where the MIMO channel is well-conditioned and linear detection methods also perform well, a-FlexCore reduces the number of active processing elements to almost one, resulting in an overall complexity similar to that of linear methods.

5.2 Algorithmic/GPU-Based Evaluation

Methodology and setup. We compare FlexCore's combined kernel execution and memory transfer times against MIMOPACK's parallel and single-threaded FCSD employing CUDA 7.5 and OpenMP. We choose MIMOPACK's FCSD over other state-of-the-art GPU implementations such as [10], since MIMOPACK is available as open-source. Implementing both detectors on the same library leads to the most fair comparison of the underlying algorithms, which is the aim of this work. We then evaluate detection performance, based on our previous assessment, in the context of the LTE standard [1]. GPU simulations are executed on the Maxwell-based GTX 970 device CPU simulations based on the OpenMP library are executed on the octacore FX-8120 x86_64 general purpose processor using 16 GB of RAM. Our profiling setups involve $|Q| \in \{16,64\}$, $N_t \in \{2,...,16\}$ and $64 \leq N_{sc} \leq 168,140$ (256 thread blocks).

FlexCore's speedup gains. Fig. 11 overlays FlexCore's

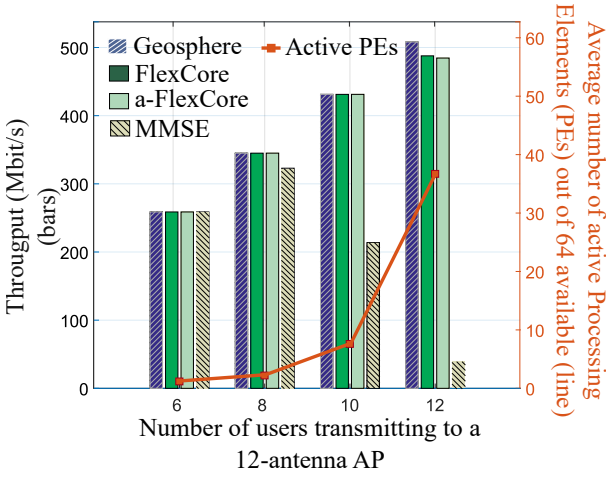


Figure 10— *Bars*: Network throughput of FlexCore and a-FlexCore with 64 available processing elements against Geosphere and MMSE, for a 12-antenna AP with six to 12 simultaneous users. *Line*: Corresponding average number of activated processing elements for a-FlexCore.

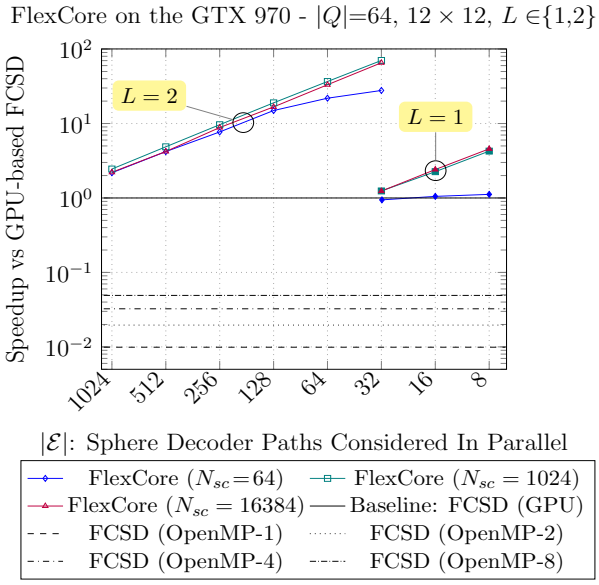


Figure 11— FlexCore's speedup vs FCSD (GPU/CPU).

speedup in 12×12 scenarios using 64-QAM modulation against the case where FCSD fully expands $L\in\{1,2\}$ levels (Sec. 2). The solid horizontal line depicts the GPU-based FCSD (baseline reference), while dashed and dotted lines display the results of CPU-based execution for a varying number of OpenMP threads (denoted as OpenMP-1, OpenMP-2 etc.). The horizontal axis lists the number of Sphere decoder tree paths considered in parallel by FlexCore.

Fig. 11 shows that the GPU-based FCSD is at least $21\times$ faster than the 8-threaded CPU version which in turn provides a maximum speedup of $5.14\times$ over single-threaded execution (*i.e.*, a 64.25% parallel efficiency). Thus, the many-core implementation and our MIMOPACK enhancements benefit both detection schemes. GPU results also show an expected sublinear increase in relative speedup as the thread ratio increases. At the SNR for which PER_{ML} is 0.01, (Fig. 9), FlexCore's speedup against the GPU-based FCSD increases up to $19\times$, as we require just 128 parallel paths to reach the same performance (Fig. 11, $L=2$). Speedup is maximized when we process in parallel a sufficient number of sphere decoder paths and/or subcarriers (*e.g.*, $N_{sc}\geq1024$ at high occupancy). When jointly assessing performance and power by employing the Joules per bit index (computed as Power (W) / Processing Throughput (bps)), FlexCore is 17.3% ($N_t=8$) up to 51.3% ($N_t=12$) more energy efficient. For $L=2$, FlexCore's energy advantage increases by 97.5%.

FlexCore for LTE. To obtain system context, we assess computation time, including data transfers, with respect to the 3GPP LTE standard [1]. LTE requires that a 10 ms frame contains 20 timeslots, each with a 500 µs duration (a total of $140\times$ the number of occupied subcarriers). Fig. 12 shows the corresponding 64-QAM SNR loss for Successive Interference Cancellation (SIC) [47], the FCSD and FlexCore compared to ML detection, based on the supported number of paths. For $N_t=8$ and when employing 8 streams, FlexCore supports 105 down to 4 paths for the two extremes of the LTE modes for which the SNR loss is 0.2 to 2.1 dB. In the $N_t=12$ case, the corresponding SNR loss becomes 0.9 to 9.8 dB (68 down to 2 paths). In the case of SIC (essentially a single-path FlexCore), the loss can be up to 11.9 dB. Notice that even though FlexCore's threads have a higher workload compared to FCSD's, the latter's inherent lack of flexibility significantly limits support to just the 1.25 MHz LTE mode for $L=1$ at $N_t\in\{8,12\}$. We note that when $L=2$, the FCSD fails to meet the LTE requirement for $N_t=8$, $|Q|=16$ and the more demanding cases.

5.3 Algorithmic/FPGA-Based Evaluation

Methodology and setup. We first present the implementation cost, achieved frequency and power consumption of a single processing element, considering $|Q|=64$, $N_t\in\{8,12\}$ at the minimum level of pipeline (Sec. 4) and 16-bits width. Then, based on the results of Sec. 5.1, we jointly evaluate power and FPGA processing throughput. Note that the number of processing elements M that can be instantiated in practice on an FPGA is limited by the latter's available resources. Due to our pipelined design though, the number of processing elements does not need to be equal to the number of Sphere decoder paths. Thus, we explore performance for varying values of M at a 5.5 ns delay (the minimum supported by both detection engines up to the number of instantiated processing elements). We note that due to host system memory lim-

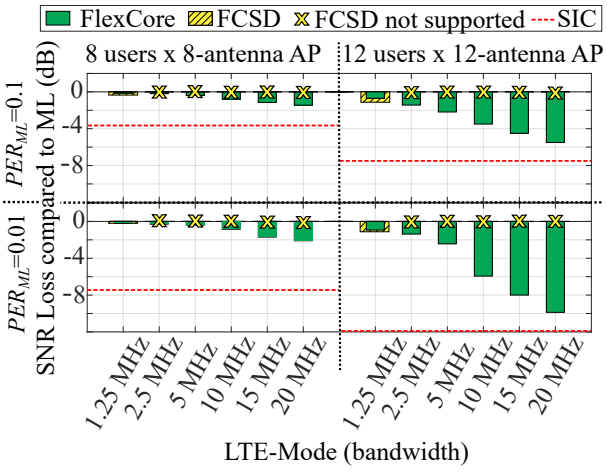


Figure 12— FlexCore, FCSD and SIC on the GPU at 64-QAM, against the ML SNR, considering the detection latency requirements of the several LTE modes.

$N_t \times N_r$	64-QAM	CLB	LUTs			DSP48	f_{max}	Power
			Logic	Mem	FF-pairs	slices	(MHz)	(W)
8×8	FlexCore	3206	15276	1187	5363	16	312.5	6.82
	FCSD	2187	11320	713	4717	16	370.4	6.54
12×12	FlexCore	5795	28810	2497	11415	24	312.5	9.157
	FCSD	4364	23252	1537	10501	24	370.4	9.04

Table 3— Single processing element on the XCVU440-flga2892-3-e for FlexCore and the FCSD at 64-QAM. FlexCore's path increases area-delay product on average only by 73.7 to 57.8% ($N_t=8$ and $N_t=12$ respectively).

itations, $M \leq 32$ in the case of FlexCore (both antenna setups) and $M \leq 32$, $M \leq 64$ respectively for the FCSD at $N_t=12$ and $N_t=8$. We estimate power through Xilinx's Power Estimator, under worst-case static power conditions at 100% utilization. Results are compared using the area-delay product and the efficiency index of Joules/bit.

FlexCore's single-path cost. Our FPGA implementation shows that FlexCore's significant advantage in terms of numbers of required processing elements (see Fig. 9), comes at a small implementation overhead per processing element. Table 3 presents implementation results for $M=1$, where a CLB is a Configurable Logic Block, containing Look-up Tables (LUTs), Flip-Flops (FFs) and distributed RAM. In fact, the processing element overhead tends to decrease as N_t increases; in the case of FlexCore and FCSD respectively for $N_t=12$, there is a 1.81× and 1.99× area delay product increase compared with the $N_t=8$ case. Timing analysis shows that logic delay is below 2 ns; the rest is attributed to routing.

Flexibility revisited: Multi-Path performance. FlexCore's design allows detection by employing an arbitrary number of Sphere decoder paths and for $M=32$, its processing throughput on this device can reach 13.09 Gbps when 32 paths need to be processed, to 3.27 Gbps (128

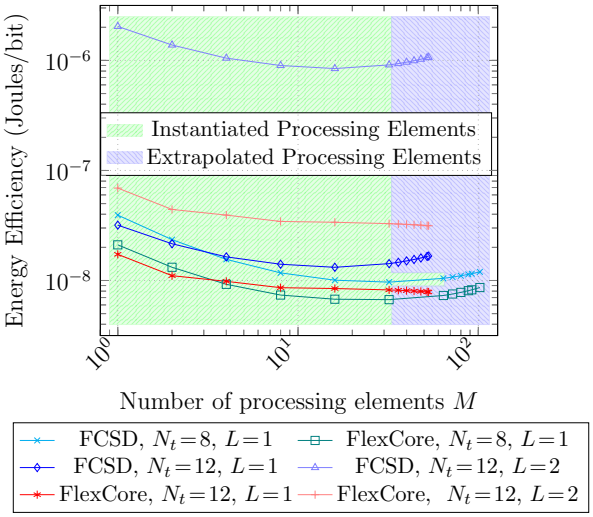


Figure 13— FPGA energy efficiency exploration on the XCVU440-flga2892-3-e under the same FlexCore and FCSD network throughput requirements (Fig. 9, $|Q|=64$).

required paths). We remind that for 12×12 MIMO, FlexCore with 32 and 128 paths reaches the same network throughput as the FCSD does with 64 and 4096 paths respectively (see Fig. 9). To support the 20 MHz LTE bandwidth for 32 and 128 Sphere decoder paths, three or more and nine or more FlexCore processing elements need to be respectively instantiated at 5.5 ns. In contrast, as the FCSD requires at least $|Q|$ PD outputs (to visit all nodes on all tree levels for $L=1$), its processing throughput is $\frac{log_2(|Q|)\cdot N_t \cdot f_{max}}{|Q|/M}$ Mbps (f_{max} is the maximum operating frequency in MHz). Even when we instantiate $M=64$ processing elements, the FCSD fails to meet the 20 MHz LTE bandwidth for $L=2$ (4096 required paths) and it requires instantiation of twice as many processing elements as FlexCore to support the 1.25 MHz mode. It is thus less area and energy efficient. Fig. 13 displays the energy efficiency of the proposed detection engines in terms of J/bit for a varying number of processing elements to reach the same network throughput (see Fig. 9). For extrapolating the numbers of processing elements that can be instantiated, a 75% maximum device utilization [3] is assumed in order to retain performance by avoiding routing congestion. Overall, the FCSD requires on average 1.54× up to 28.8× more J/bit ($N_t=8$, $L=1$ and $N_t=12$, $L=2$ cases respectively). These results illustrate FlexCore's significant efficiency advantage. To put GPU performance into perspective, our FPGA implementations achieve a two and three orders of magnitude higher energy efficiency for $L=1$ and $L=2$, respectively.

Discussion. We have shown that FlexCore can efficiently exploit any number of available processing elements, and can outperform linear detection methods even when a small number of processing elements is available. Compared to the FCSD, FlexCore can provide significant

throughput gains for the same number of processing elements and it can achieve near-optimal performance with a number of processing elements that can be more than an order of magnitude less. FlexCore's modest detection requirements translate to a GPU implementation speedup of 19× (64-QAM, 12×12 MIMO at 21.6 dB) against the FCSD when the latter fully expands two levels (Fig. 9 and 11). FlexCore's flexibility allows it to operate in varying conditions, in which the FCSD fails to meet the LTE's timing requirements. To the best of our knowledge, FlexCore is the first Sphere-decoding-based detection scheme that can support all LTE bandwidths, while providing performance better than the one of SIC, even for 12×12 MIMO systems on a GPU [33, 35, 40, 50, 51]. Our FPGA implementation results reveal that FlexCore is 34% to 96% more energy efficient than the proposed FCSD implementation and can reach a throughput of 13 Gbps on a state-of-the-art device. Also to the best of our knowledge, these are the first FCSD-based FPGA implementations for 8×8 and larger antenna arrays [2, 5, 26, 49].

6 Related Work

The exploitation of parallelism alone to reduce processing latency is not a novel approach: indeed, implementations of all kinds of Sphere decoders (e.g., both breadth-first and depth-first) involve some level of parallelism. However, existing approaches either take a limited or inflexible level of parallelism or they perform parallel processing takes place in an suboptimal, heuristic manner, without accounting for the actual transmission channel conditions.

Parallelism at a distance calculation level. Both depth-first [8, 20, 45] and breadth-first Sphere decoder implementations, including the *K-Best sphere decoders* [9, 18, 28, 30, 36, 37, 46, 48], calculate multiple Euclidean distances in parallel any time they change tree level. In addition, after performing the parallel operations, the node or list of nodes with the minimum Euclidean distance needs to be found, which requires a significant synchronization overhead between the parallel processes. This level of exploited parallelism is fixed, predetermined, non-flexible, with high dependencies which are related to the specific architectural design. In addition, in K-Best Sphere decoders the value of K, which is predetermined, needs to increase for dense constellations and large numbers antennas, making K-best detection inappropriate for dense constellations and large MIMO systems. Using FlexCore's approach we can adaptively select the value of K, which will differ per Sphere decoding tree level.

Parallelism at a higher than a Euclidean distance level. Khairy *et al.* [27] use GPUs to run in parallel multiple, low-dimensional (4×4) Sphere decoders but without parallelizing the tasks or the data processing involved in each. Jósza *et al.* [25] describe a hybrid *ad hoc* depth-first, breadth-first GPU implementation for low dimensional sphere decoders. However, their approach lacks theoretical basis and cannot prevent visiting unnecessary tree paths that are not likely to include the correct solution. In addition, since the authors do not propose a specific tree search methodology, their approach is not extendable to large MIMO systems. Yang *et al.* [52, 53] propose a VLSI/CMOS multicore combined distributed/shared memory approach for high-dimensional SDs, where SD partitioning is performed by splitting the SD tree into subtrees. But partitioning is heuristic, and their approach requires interaction between the parallel trees, thus making it inflexible. In addition, the required communication overhead among the parallel elements makes the approach inefficient for very dense constellations and inappropriate for a GPU implementation.

Other detection approaches. Local area search approaches [29,39,42] could also be used for the detection of large MIMO systems, but they are strictly sequential and require several iterations, resulting in increased latency. Lattice reduction techniques [15] are also prohibitive for large MIMO systems due to their sequential manner and high complexity ($\mathcal{O}(N_t^4)$).

7 Conclusion and Future Work

We have described FlexCore, a computationally-flexible method to consistently and massively parallelize the problem of detection in large MIMO systems, as well as similar maximum-likelihood detection problems. Our FlexCore GPU implementation in 12×12 64-QAM MIMO, enjoys a 19× computational speedup and 97% increased energy efficiency compared with the state-of-the-art. Furthermore, according to the best of our knowledge, our FlexCore's GPU implementation is the first able to support all LTE bandwidths and provide detection performance better than SIC, even for 12×12 MIMO systems. Finally, our implementation and exploration of FlexCore on FPGAs showed that for the same MIMO system, its energy efficiency surpasses that of the state-of-the-art by an order of magnitude. A promising next step is to extend FlexCore to "soft-detectors" as in [7, 43].

8 Acknowledgments

This material is based upon work supported by the UK's Engineering and Physical Sciences Research Council (EPSRC Ref. EP/M029441/1), by the National Science Foundation under Grant No. 1617161 and the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement No. 279976. The Authors would like to thank the members of University of Surrey 5GIC (http://www.surrey.ac.uk/5GIC) for their support. Finally, we thank our shepherd Lin Zhong and the anonymous reviewers for their insightful feedback.

References

[1] 3GPP LTE Encyclopedia: An Introduction to LTE, 2010.

[2] K. Amiri, C. Dick, R. Rao, J. R. Cavallaro. Flex-sphere: An FPGA configurable sort-free sphere detector for multi-user MIMO wireless systems. *Software Defined Radio Forum (SDR)*, 2008.

[3] D. Amos, A. Lesea, R. Richter. *FPGA-based prototyping methodology manual: best practices in design-for-prototyping*. Synopsys Press, 2011.

[4] L. Barbero, J. Thompson. Fixing the complexity of the sphere decoder for MIMO detection. *IEEE Transactions on Wireless Communications*, **7**(6), 2131–2142, 2008.

[5] L. G. Barbero, J. S. Thompson. FPGA design considerations in the implementation of a fixed-throughput sphere decoder for MIMO systems. *IEEE International Conference on Field Programmable Logic and Applications*, 1–6, 2006.

[6] J. R. Barry, E. A. Lee, D. G. Messerschmitt. *Digital Communication*. Springer Science & Business Media, 2004.

[7] J. Boutros, N. Gresset, L. Brunel, M. Fossorier. Soft-input soft-output lattice sphere decoder for linear channels. *IEEE Global Telecommunications Conference (GLOBECOM)*, vol. 3, 1583–1587, 2003.

[8] A. Burg, M. Borgmann, M. Wenk, M. Zellweger, W. Fichtner, H. Bolcskei. VLSI implementation of MIMO detection using the sphere decoding algorithm. *IEEE Journal of Solid-State Circuits*, **40**(7), 1566–1577, 2005.

[9] S. Chen, T. Zhang, Y. Xin. Relaxed K-best MIMO signal detector design and VLSI implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **15**(3), 328–337, 2007.

[10] T. Chen, H. Leib. GPU acceleration for fixed complexity sphere decoder in large MIMO uplink systems. *2015 IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE)*, 771–777, 2015.

[11] Chia-Hsiang Yang, D. Markovic. A flexible DSP architecture for MIMO sphere decoding. *IEEE Transactions on Circuits and Systems I: Regular Papers*, **56**(10), 2301–2314, 2009.

[12] R. Courtland. Transistors could stop shrinking in 2021. *IEEE Spectrum*, **53**(9), 9–11, 2016.

[13] V. K. D Wübben, J Rinas, K. Kammeyer. Efficient algorithm for decoding layered space-time codes. *IEEE Electronics letters*, **37**(22), 1348–1350, 2001.

[14] E. Viterbo, and E. Biglieri. A universal decoding algorithm for lattice codes. *Proceedings of GRETSI*, 611–614. Juan-les-Pins, France, 1993.

[15] Y. H. Gan, C. Ling, W. H. Mow. Complex lattice reduction algorithm for low-complexity full-diversity MIMO detection. *IEEE Transactions on Signal Processing*, **57**(7), 2701–2710, 2009.

[16] G. Georgis, G. Lentaris, D. Reisis. Acceleration techniques and evaluation on multi-core CPU, GPU and FPGA for image processing and super-resolution. *Journal of Real-Time Image Processing*, 1–28, 2016.

[17] R. E. Guerra, N. Anand, C. Shepard, E. W. Knightly. Opportunistic channel estimation for implicit 802.11 af MU-MIMO. *Teletraffic Congress (ITC 28), 2016 28th International*, vol. 1, 60–68. IEEE, 2016.

[18] Z. Guo, P. Nilsson. Algorithm and implementation of the K-best sphere decoding for MIMO detection. *IEEE Journal on Selected Areas in Communications*, **24**(3), 491–503, 2006.

[19] B. Hassibi, H. Vikalo. On the sphere-decoding algorithm I. Expected complexity. *IEEE Transactions on Signal Processing*, **53**(8), 2806–2818, 2005.

[20] C. Hess, M. Wenk, A. Burg, P. Luethi, C. Studer, N. Felber, W. Fichtner. Reduced-complexity MIMO detector with close-to ML error rate performance. *Proceedings of the 17th ACM Great Lakes symposium on VLSI*, 200–203. ACM, 2007.

[21] IEEE Standard for Information Technology, part 11: wireless LAN Medium Access Control (MAC) and physical layer (PHY) specifications, amendment 4: enhancements for very high throughput for operation in bands below 6 GHz (IEEE Std 802.11ac-2013).

[22] IEEE Standard 802.11: wireless LAN medium access control and physical layer specifications, 2012.

[23] Intel's Skylake Core i7: a performance look, 2016. http://techgage.com/article/intels-skylake-core-i7-6700k-a-performance-look.

[24] J. Jaldén, B. Ottersten. On the complexity of sphere decoding in digital communications. *IEEE Transactions on Signal Processing*, **53**(4), 1474–1484, 2005.

[25] C. M. Józsa, G. Kolumbán, A. M. Vidal, F.-J. Martínez-Zaldívar, A. González. New parallel sphere detector algorithm providing high-throughput for optimal MIMO detection. *Procedia Computer Science*, **18**, 2432 – 2435, 2013.

[26] M. S. Khairy, M. M. Abdallah, S.-D. Habib. Efficient FPGA implementation of MIMO decoder for mobile WiMAX system. *IEEE International Conference on Communications*, 1–5, 2009.

[27] M. S. Khairy, C. Mehlführer, M. Rupp. Boosting sphere decoding speed through Graphic Processing

Units. *European Wireless Conference (EW)*, 99–104, 2010.

[28] Q. Li, Z. Wang. Improved K-best sphere decoding algorithms for MIMO systems. *IEEE International Symposium on Circuits and Systems*, 1159–1162, 2006.

[29] S. K. Mohammed, A. Chockalingam, B. S. Rajan. A low-complexity near-ML performance achieving algorithm for large MIMO detection. *IEEE International Symposium on Information Theory*, 2012–2016, 2008.

[30] S. Mondal, A. Eltawil, C. A. Shen, K. N. Salama. Design and implementation of a sort-free K-best sphere decoder. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **18**(10), 1497–1501, 2010.

[31] J. Nickolls, I. Buck, M. Garland, K. Skadron. Scalable parallel programming with CUDA. *ACM Queue Magazine*, **6**(2), 40–53, 2008.

[32] K. Nikitopoulos, J. Zhou, B. Congdon, K. Jamieson. Geosphere: Consistently turning MIMO capacity into throughput. *Proceedings of the ACM SIGCOMM*, 631–642. ACM, 2014.

[33] T. Nyländen, J. Janhunen, O. SilvÃl'n, M. Juntti. A GPU implementation for two MIMO-OFDM detectors. *IEEE International Conference on Embedded Computer Systems (SAMOS)*, 293–300, 2010.

[34] C. Ramiro, A. M. Vidal, A. Gonzalez. MIMOPack: a high-performance computing library for MIMO communication systems. *The Journal of Supercomputing*, **71**(2), 751–760, 2014.

[35] S. Roger, C. Ramiro, A. Gonzalez, V. Almenar, A. M. Vidal. Fully parallel GPU implementation of a fixed-complexity soft-output MIMO detector. *IEEE Transactions on Vehicular Technology*, **61**(8), 3796–3800, 2012.

[36] M. Shabany, P. G. Gulak. Scalable VLSI architecture for K-best lattice decoders. *IEEE International Symposium on Circuits and Systems*, 940–943, 2008.

[37] M. Shabany, K. Su, P. G. Gulak. A pipelined scalable high-throughput implementation of a near-ML K-best complex lattice decoder. *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 3173–3176, 2008.

[38] C. Shepard, H. Yu, N. Anand, L. Li, T. Marzetta, R. Yang, L. Zhong. Argos: Practical many-antenna base stations. *Proceedings of ACM Conference on Mobile Computing and Networking (MobiCom)*, 2012.

[39] N. Srinidhi, S. K. Mohammed, A. Chockalingam, B. S. Rajan. Near-ML signal detection in large-dimension linear vector channels using reactive tabu search. *arXiv preprint arXiv:0911.4640*, 2009.

[40] D. Sui, Y. Li, J. Wang, P. Wang, B. Zhou. High throughput MIMO-OFDM detection with Graphics Processing Units. *IEEE International Conference on Computer Science and Automation Engineering (CSAE)*, vol. 2, 176–179, 2012.

[41] K. Tan, *et al.* SAM: Enabling practical spatial multiple access in wireless LAN. *Proceedings of ACM Conference on Mobile Computing and Networking (MobiCom)*, 2009.

[42] K. V. Vardhan, S. K. Mohammed, A. Chockalingam, B. S. Rajan. A low-complexity detector for large MIMO systems and multicarrier CDMA systems. *IEEE Journal on Selected Areas in Communications*, **26**(3), 473–485, 2008.

[43] H. Vikalo, B. Hassibi, T. Kailath. Iterative decoding for MIMO channels via modified sphere decoding. *IEEE Transactions on Wireless Communications*.

[44] E. Viterbo, J. Boutros. A universal lattice code decoder for fading channels. *IEEE Transactions on Information Theory*, **45**(5), 1639–1642, 1999.

[45] M. Wenk, L. Bruderer, A. Burg, C. Studer. Area-and throughput-optimized VLSI architecture of sphere decoding. *IEEE/IFIP 18th VLSI System on Chip Conference (VLSI-SoC)*, 189–194, 2010.

[46] M. Wenk, M. Zellweger, A. Burg, N. Felber, W. Fichtner. K-best MIMO detection VLSI architectures achieving up to 424 Mbps. *IEEE International Symposium on Circuits and Systems*, 4 pp.–1154, 2006.

[47] P. W. Wolniansky, G. J. Foschini, G. Golden, R. A. Valenzuela. V-BLAST: An architecture for realizing very high data rates over the rich-scattering wireless channel. *IEEE URSI International Symposium on Signals, Systems, and Electronics (ISSSE)*, 295–300, 1998.

[48] K. wai Wong, C. ying Tsui, R. S. K. Cheng, W. ho Mow. A VLSI architecture of a K-best lattice decoding algorithm for MIMO channels. *IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 3, III–273–III–276, 2002.

[49] B. Wu, G. Masera. A novel VLSI architecture of fixed-complexity sphere decoder. *13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD)*, 737–744, 2010.

[50] M. Wu, S. Gupta, Y. Sun, J. R. Cavallaro. A GPU implementation of a real-time MIMO detector. *IEEE Workshop on Signal Processing Systems*, 303–308, 2009.

[51] M. Wu, Y. Sun, S. Gupta, J. R. Cavallaro. Implementation of a high throughput Soft MIMO

Detector on GPU. *Journal of Signal Processing Systems*, **64**(1), 123–136, 2011.

[52] C. H. Yang, D. Marković. A 2.89mW 50GOPS 16x16 16-core MIMO sphere decoder in 90nm CMOS. *IEEE European Solid-State Circuits Conference (ESSCIRC)*, 344–347, 2009.

[53] C. H. Yang, T. H. Yu, D. Marković. A 5.8mW 3GPP-LTE compliant 8x8 MIMO sphere decoder chip with soft-outputs. *IEEE Symposium on VLSI Circuits (VLSIC)*, 209–210, 2010.

[54] Q. Yang, X. Li, H. Yao, J. Fang, K. Tan, W. Hu, J. Zhang, Y. Zhang. Bigstation: enabling scalable real-time signal processing in large MU-MIMO systems. *ACM SIGCOMM Computer Communication Review*, **43**(4), 399–410, 2013.

A Position Vector Error Probability Approximation

For the top sphere decoding tree layer ($l = N_t$), the probability of the first closest symbol to the effective received point $\tilde{y}(N_t)$ not to be the transmitted symbol is equivalent to the corresponding symbol error rate over an AWGN channel, or [6]

$$P_e(N_t) = \left(2 + \frac{2}{\sqrt{|Q|}}\right) \cdot \text{erfc}\left(\frac{R(N_t,N_t)\cdot\sqrt{Es}}{\sigma}\right), \quad (6)$$

Then, the probability of the first closest symbol to the received to be the transmitted one is $P_{N_t}(1) = 1 - P_e(N_t)$. Calculating the probability for the kth (with $k > 1$) closest to the received symbol to be the one transmitted would require real-time two-dimensional integrations since an analytical solution is infeasible. Instead, we approximate the problem based on the observation that the inter-symbol distance in QAM constellations scales nearly in a square-root manner, as a function of the position index k related to the received signal.

Then we make the approximation that the decision boundaries (D_k) would scale in a similar manner. That is

$$D_k = \sqrt{c \cdot k}, \quad (7)$$

where c is a positive and real constant. Then,

$$P_{N_t}(k) = P\left(D_{k-1} < |n_{N_t}| \leq D_k\right)$$

$$= P\left(|n_{N_t}| \leq \sqrt{c\cdot(k)}\right) - P\left(|n_{N_t}| \leq \sqrt{c\cdot(k-1)}\right). \quad (8)$$

Since the amplitude of the noise sample (n_{N_t}) is Rayleigh distributed

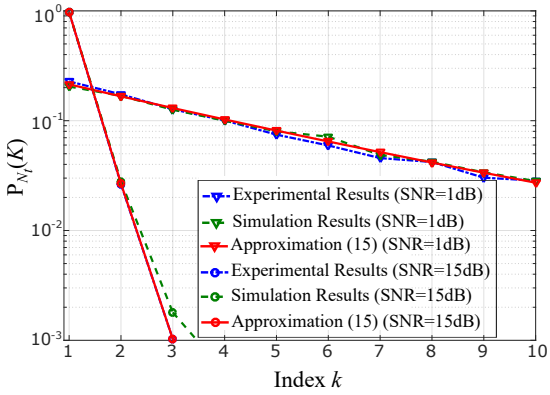


Figure 14— Comparison between (11) (solid), simulation (dashed, Gaussian noise) results for $P_{N_t}(k)$, and experimental results (dashed-dotted, WARP platform) for the probabilities $P_{N_t}(k)$ at various SNRs.

$$P_{N_t}(k) = \exp\left(-\frac{c\cdot(k-1)}{\sigma_{n_t}^2}\right) - \exp\left(-\frac{c\cdot(k)}{\sigma_{n_t}^2}\right)$$
$$= \exp\left(-\frac{c\cdot(k-1)}{\sigma_{n_t}^2}\right)\cdot\left[1 - \exp\left(-\frac{c}{\sigma_{n_t}^2}\right)\right] \quad (9)$$

Applying the above for $k = 1$ is $1 - P_e(N_t)$, with $P_e(N_t)$ defined in (6), therefore, for both equations to hold,

$$P_e(N_t) = \exp\left(-\frac{c}{\sigma_l^2}\right). \quad (10)$$

Accordingly, the probability that the kth closest constellation point to the observable of the top level ($l = N_t$) is the transmitted one can be expressed as

$$P_{N_t}(k) = (1 - P_e(N_t))\cdot(P_e(N_t))^{(k-1)}. \quad (11)$$

Fig. 14 compares the theoretical estimates of the "per-level" probabilities P_{N_t} to the ones obtained by simulations as well as to the ones obtained by actual experiments using our WARP v3 platform implementation (described in Section 5.1). It shows that our theoretical model is very accurate in all SNR regimes.

It can be easily shown that, the above equation does hold for any sphere decoding tree level, given that all the higher layers include the correct solutions (the correct transmitted vector). This is because the effect of the correct solution can be easily removed in terms of successive interference cancellations. As as result, the probability P_c can be calculated as in equations (2), (3) and (4) (Sec. 3).

Facilitating Robust 60 GHz Network Deployment By Sensing Ambient Reflectors

Teng Wei†, Anfu Zhou* and Xinyu Zhang†
†University of Wisconsin-Madison, *Beijing University of Posts and Telecommunications
twei7@wisc.edu, zhouanfu@gmail.com, xyzhang@ece.wisc.edu

Abstract

60 GHz millimeter-wave networks represent the next frontier in high-speed wireless access technologies. Due to the use of highly directional and electronically steerable beams, the performance of 60 GHz networks becomes a sensitive function of environment structure and reflectivity, which cannot be handled by existing networking paradigms. In this paper, we propose E-Mi, a framework that harnesses 60 GHz radios' sensing capabilities to boost network performance. E-Mi uses a single pair of 60 GHz transmitter and receiver to sense the environment. It can resolve all dominant reflection paths between the two nodes, from which it reconstructs a coarse outline of major reflectors in the environment. It then feeds the reflector information into a ray-tracer to predict the channel and network performance of arbitrarily located links. Our experiments on a custom-built 60 GHz testbed verify that E-Mi can accurately sense a given environment, and predict the channel quality of different links with 2.8 dB median error. The prediction is then used to optimize the deployment of 60 GHz access points, with $2.2\times$ to $4.5\times$ capacity gain over empirical approaches.

1. Introduction

The unlicensed millimeter wave (mmWave) band, centered at 60 GHz and spanning 14 GHz spectrum [1], represents the most promising venue to meet the massive surge in mobile data. Recently proposed mmWave network standards, like 802.11ad [2], provision multi-Gbps connectivity for a new wave of applications such as cordless computing and wireless fiber-to-home.

Despite the huge potential, 60 GHz networks face a number of challenges unseen in conventional low-frequency networks: due to ultra-high carrier frequency, the 60 GHz radios are extremely vulnerable to propagation loss and obstacle blockage. To overcome such limitation, 60 GHz radios commonly adopt many-element *phased-array antennas* to form highly directional, steerable beams, which leverage reflections to steer around obstacles. Dependence on directivity and reflection, however, makes the network performance a sensitive function of node placement and environmental characteristics (*e.g.*, geometrical layout and reflectivity of ambient surfaces).

To elucidate the challenge, we set up two laptops with Qualcomm tri-band QCA6500 chip (2.4/5/60GHz) [3], and randomly place them over 16 different locations. For each link, we measured the normalized throughput, *i.e.*, the none-line-of-sight (NLOS) throughput when a human obstacle stands in between, divided by the LOS throughput. Fig. 1 plots the CDF across links. We observe up to $3\times$ throughput gap when the *same-distance* 60 GHz link is placed at different locations, versus $1.4\times$ for the 2.4 GHz link, implying that 60 GHz's NLOS performance is much more sensitive to environment. We found that the 60 GHz transmitter can more effectively detour blockage, if it is placed near a concrete wall that acts like a mirror. Obviously, reflectors in the environment have a crucial impact on 60 GHz performance.

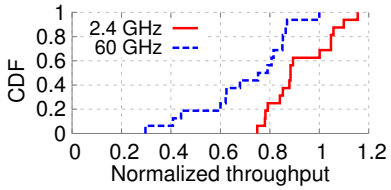


Figure 1: Normalized throughput of same-distance 60 GHz and 2.4 GHz links over different node placements.

Of course, one may not always be able to alter the environment. However, we argue that, by judiciously placing 60 GHz access points (APs) within a given environment, we can substantially improve network coverage and robustness to blockage. To this end, one may conduct a blanket site-survey and search for the capacity-maximizing AP location, but the search space becomes formidable because of the numerous beam directions and human blockage patterns. In this paper, we propose *E-Mi,* a system that can automatically "sense" (model) the major reflectors in the environment from a 60 GHz radio's eyes, and predict the performance of arbitrarily located links. The prediction can in turn help optimize AP placement to maximize network capacity and robustness.

The core challenge in E-Mi is: how to sense the environment using mmWave radios which can only measure the received signal strength (RSS) and phase between each other? Conventional environment mapping approaches (*e.g.*, stereo camera and laser radar [4–8]) need dedicated hardware and do not capture environment properties specific to mmWave. In contrast, E-Mi leverages the well known sparsity of mmWave channels [9–11]: from the 60 GHz radios' eyes, there are usually only a few *dominating reflectors* in practical environment. E-Mi samples the environment by fixing the Tx radio, and

moving the Rx to a few locations. At each location, the radio channel comprises one LOS path, and many NLOS ones. By measuring the RSS/phase, E-Mi traces back all NLOS propagation paths, uses a geometrical model to locate where the paths hit reflectors, and eventually reverse-engineers the location and reflectivity of dominating reflectors. Such environment information is then fed into a ray tracing engine, which can predict the wireless channel quality of arbitrarily located Tx/Rx.

E-Mi's reflector learning is predicated on the accurate tracing of propagation paths, which itself is an open challenge. Specifically, E-Mi needs to disentangle all the NLOS paths for each Rx location, and estimate each path's angle of arrival (AoA), angle of departure (AoD) and length. This differs from the vast literature of phased-array localization algorithms that only exploit the LOS path [12, 13]. E-Mi solves the problem using a *multi-path resolution framework* (MRF), which resolves different paths' angles/lengths by creating "virtual beams" by post-processing the measured RSS/phase.

We have implemented E-Mi on a 60 GHz testbed. Our experiments demonstrate that E-Mi can accurately resolve NLOS propagation paths, with an average error of 3.5°, 3.5°, and 0.4 m, for AoA, AoD and path length, respectively. By simply sampling 15 receiver locations in an office environment, E-Mi can effectively predict the link quality of other unobserved locations, with median RSS error of 2.8 dB and AoA(AoD) error $4.5^\circ(5.7^\circ)$.

E-Mi can be a convenient toolset to predict site-specific RSS distributions and assist 60 GHz network deployment and configuration. In this paper, we apply E-Mi to one case study to answer the following question: How to deploy the 60 GHz APs to maximize the average network capacity and improve resilience to blockage? Our experiments show that an E-Mi-augmented deployment obtains $2.2\times$ to $4.5\times$ median throughput gain over empirical approaches. E-Mi also makes the 60 GHz network more robust, reducing median throughput loss from around 700 Mbps to 20 Mbps under random human blockage.

To summarize, the main contributions of E-Mi include:

(i) A multipath resolution framework that allows a pair of 60 GHz Tx and Rx to trace back the $\langle$*AoA, AoD, length*$\rangle$ of all NLOS paths, simply via RSS/phase measurement.

(ii) An reflector localization scheme that can locate where the reflectors "bend" propagation paths, and then recover the layout/reflectivity of dominant reflectors.

(iv) Applying the sensing information to predicting the channel quality of arbitrarily located Tx and Rx, which in turn helps optimize the AP deployment, achieving multi-fold capacity gain and robustness under human blockage.

2. Related Work

Wireless network planning/profiling. Wireless network planning is a classical problem that has been relying on empirical solutions for decades. RF site survey, despite its tedious war-driving procedure, is still widely adopted by enterprise WLAN and cellular network planning tools [14, 15]. Recent work used roaming robots [16] or sparse sampling [17] to access the RSS distribution under a given AP/basestation deployment. But these approaches hardly shed lights on how to plan a new/better deployment.

Active planning can overcome the limitation by using ray-tracing. Earlier study of 60 GHz channel statistical characterization [18] unveiled that 60 GHz signals have predictable spatial structure in an environment. But they require precise mapping of dominant reflectors. Such a map is not always available and is sensitive to environmental change (*e.g.*, placing a new cabinet). E-Mi essentially circumvents this hurdle by allowing mmWave radios to directly construct the environment map.

Radio-based environment sensing. The simultaneous localization and mapping (SLAM) problem has been extensively studied in robotics [19–21]. Typical SLAM systems need to roam a robot, and map the environment based on dead-reckoning and visual images. Such systems are predicated on two factors: *(i)* precisely controlled robotic movement and blanket coverage, to generate an extensive point-cloud representation of the survey area. *(ii)* environment sensors, such as sonar, stereo camera and LIDAR [22], to explicitly locate landmarks or obstacles. The elusive nature of wireless signals prohibits us from meeting the same requirement. It involves nontrivial human efforts to label the reflectivity of each reflector. Besides, the reflectivity may be hardly available if an object contains compound materials. State-of-the-art radio-based SLAM [5,23] can only achieve localization accuracy of around 5 meters, far from enough to predict the spatial performance of a wireless network.

Recent work [7,24,25] adopted mmWave radars to explicitly *scan* the environment. By continuously moving the radar in front of the obstacle's body, they can identify the shape/reflectivity. In contrast, E-Mi leverages the sparsity of 60 GHz signal structure, so as to locate all dominant reflectors with only a few sampling locations.

Localization using antenna arrays. Antenna array has demonstrated tremendous potential in localization, especially because it can identify AoA using signal processing algorithms like MUSIC [26] and ESPRIT [27]. Recent systems [12, 13, 28] renovated such algorithms to localize a client via multi-AP triangulation. In contrast, E-Mi uses mmWave phased-array to handle the more challenging problem of recovering NLOS propagation paths and locating reflectors. A side benefit of E-Mi is that it can locate a node using a single AP (Sec. 5) and therefore build a spatial distribution map of possible client locations, which can in turn help optimize the AP deployment (Sec. 9).

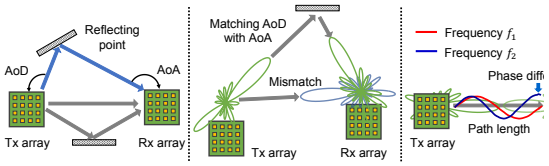


Figure 2: The MRF identifies the ⟨AoA, AoD, le in three steps: (1) Identify the dominant AoA and (2) Pair AoA and AoD directions that belong to the path; (3) Estimate the length of each path.

Sensor-assisted protocol adaptation. E-Mi is i inspired by the principle of sensor-assisted commu tions. Nanthapuri *et al.* [29] proposed to discrin various networking context (*e.g.*, mobile vs. static) using external sensors, and adapt the protocols accordingly. Ravindranath *et al.* [30] applied a similar principle to assisting link-level rate adaptation, *etc.*. BBS [31] leveraged a WiFi antenna array to estimate the signal's AoA and facilitate the 60 GHz radio beam steering. BeamSpy [32] detects human blockage and adapts its beam to a new direction without beam scanning. In contrast to this line of research, E-Mi uses 60 GHz radios themselves as sensors to reconstruct the reflectors and predict the site-specific RSS distribution to guide AP deployment.

3. E-Mi: An Overview

E-Mi samples the RSS/phase between a pair of 60 GHz AP and client (also denoted as Tx and Rx), and uses the samples as input to two major modules: *(i) Multi-path resolution framework (Sec. 4)*, which estimates the geometry, *i.e.*, ⟨*AoA, AoD, length*⟩, of each propagation path and also discriminates their RSS/phase. *(ii) Dominant reflector reconstruction (Sec. 5): which locates the reflecting points (i.e., spots where the paths hit the reflector)*, and reconstructs the layout/reflectivity of dominating reflectors, forming a coarse environment map. A network planner can feed E-Mi's reconstruction result to a 60 GHz ray-tracing engine, and identify the AP locations that lead to higher capacity/robustness (Sec. 9). This essentially supersedes the laborious war-driving in traditional wireless site survey [15].

E-Mi requires the Tx and Rx to be equipped with phased-arrays of practical size (default to 16-element, as in typical 802.11ad radios [33]). It does not need a customized PHY layer—It only requires the channel state, which is a portable function on many commodity WiFi devices [34] and expected to be available in the 802.11ad products. Although E-Mi works in a constrained environment that can be illuminated by Tx's signals, the Tx can be moved to different positions to extend its coverage.

When scanning the wireless channel, E-Mi places the Tx and Rx well above the ground, so that they only "see" dominant reflectors like walls and furnitures. They affect the average-case network performance which are of utmost interest for network planners. In case such reflectors change their locations, we can accommodate the changes by rerunning E-Mi.

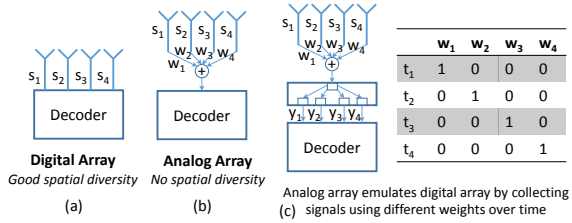


Figure 3: Isolating individual antenna's signals on an analog phased array, by switching across different weight vectors.

4. Multipath Resolution Framework (MRF)

The MRF estimates the ⟨*AoA, AoD, length*⟩ of dominating paths between the Tx and Rx. As illustrated in the Fig. 2, the AoA and AoD are determined by the relative positions of dominant reflector, Tx and Rx, and independent of the beam pattern of phased arrays. To estimate these intrinsic parameters, a naive solution is to use *beam scanning*: the Tx/Rx may steer over all possible combinations of beam directions, and find the ones with high RSS. However, a 60 GHz phased-array can only steer to a set of discrete directions (*e.g.*, a 16-element one can only steer between beams with 22.5° separation [35]). The discrete beam scanning prevents us from measuring the signal angle precisely. Moreover, unlike horn antennas, phased-arrays have imperfect directionality — besides the main beam, their antenna pattern bears many sidelobes which interfere the AoA/AoD estimation.

E-Mi's MRF introduces three mechanisms (Fig. 2) to meet the challenges. *(i)* We first estimate the dominating AoD and AoA directions, originating from Tx and ending at Rx, respectively. We adapt a classical signal angle estimation algorithm to 60 GHz phased arrays, which enables super-resolution (*i.e.*, finer resolution than discrete steering by generating a continuous angular spectrum, and unaffected by the imperfect beam shape of phased arrays). *(ii)* We design a *virtual beamforming* (vBeam) scheme that pairs the AoD and AoA directions belonging to the same NLOS path. *(iii)* We employ a *multi-tone ranging* scheme to estimate the total length of each path.

4.1 Estimate Path Angles Using Phased Arrays

Conventional multi-antenna receivers can estimate signal AoA using *angular spectrum analysis*, which singles out the arrival angles with strong signal strength [12, 13]. However, such analysis needs to *isolate* the signals on each antenna element using digital phased arrays (Fig. 3(a)). Practical 60 GHz radios use *analog phased arrays* (Fig. 3(b)), which have a single input/output, comprising a weighted sum of individual antenna's signals that obfuscate each antenna' signals.

To overcome this limitation, a natural way is to vary the weights and obtain a system of equations to solve for individual signals. Suppose there are N elements on the receiving phased array, and $\mathbf{S} = [S_1, S_2, \cdots, S_N]^T$ denotes the signals on the N individual antenna elements. When the phased array imposes row vector of weight $\mathbf{w}_1$ on its antenna elements, the received signal becomes $y_1 = \mathbf{w}_1\mathbf{S}$. Suppose the array switches across N different weight vectors to receive the same signals by N times. The weights constitute a 2-D matrix $\mathbf{W}_R = [\mathbf{w}_1; \mathbf{w}_2; \cdots; \mathbf{w}_N]$ with each row being a weight vector. Then, the N output signals form a vector: $\mathbf{Y} = \mathbf{W}_R\mathbf{S}$. So, one can simply use $\mathbf{S} = \mathbf{W}_R^{-1}\mathbf{Y}$ to recover $\mathbf{S}$, and hence isolate the signals on each antenna element. Fig. 3(c) illustrates an example where $N = 4$.

In practice, the weight vectors in a 60 GHz phased array are built into hardware and can only be selected from a predefined group, called *codebook*. The key question is: can we find a set of weight vectors to form a matrix $\mathbf{W}_R$ that is invertible? The answer is positive: we can find the weight matrix from 60 GHz codebook that is orthogonal (*i.e.*, $\mathbf{W}_R\mathbf{W}_R^H = I$, where $(\cdot)^H$ denotes the conjugate transpose), and hence invertible. The beamforming codebook ensures orthogonality between weight vectors because it will maximize the isolation across different beam patterns [35, 36].

To estimate the AoD, a symmetrical operation is needed at the Tx. Suppose the Tx phased array has M antenna elements, then it uses M different sets of weights to transmit the signals by M times, which similarly constitute a transmit matrix $\mathbf{W}_T$. We populate $\mathbf{S}$ into an N-by-M matrix. Each element (i, j) of $\mathbf{S}$ represents the signals on i-th Rx antenna, when the j-th Tx antenna element is triggered. Then, the received signals of analog-array becomes: $\mathbf{Y} = \mathbf{W}_R\mathbf{S}\mathbf{W}_T^H$. Each column/row in matrix $\mathbf{Y}$ contains received signals measured using a specific transmitting/receiving weight vector. By way of a similar orthogonality argument as the Rx, we can recover $\mathbf{S}$ as follows: $\mathbf{S} = \mathbf{W}_R^H\mathbf{Y}\mathbf{W}_T$.

Isolation of individual antenna's signals allows us to apply MUSIC [37], an eigen angle analysis algorithm to jointly estimate the AoA/AoD, in the same way as in digital phased arrays [12]. MUSIC can achieve a scalable resolution with more antenna elements and extricate the discrete beam shape of phased arrays. Specifically, we measure the preamble signals sent/received by standard 60 GHz radios [2], which are sent repeatedly across packets, and across different Tx/Rx beam patterns. We isolate the preamble signals sent/received by different Tx/Rx antenna elements. Then, we run MUSIC to compute the angular spectrum, essentially the likelihood of signals coming from different angles. Finally, we find the peaks in the angular spectrum that are larger than the noise floor and take the corresponding directions as AoAs/AoDs of dominating paths.

A few additional operations are worth noting: *(i)* Since each AoD is pairwise to an AoA *w.r.t.* the same dominant reflector, we remove the excessive AoA/AoD estimations of smaller eigenvalue, and make sure the number of AoA and AoD values are equal. *(ii)* To ensure the consistency of reference direction, *i.e.*, 0 degree, in the measurement, the antenna's orientation can be simply kept at a fixed direction, or be compensated by motion sensors in the mobile device. *(iii)* MUSIC is adopted only for AoA and AoD estimation. The RSS estimation and AoA/AoD pairing of each path is accomplished by the virtual beamforming, which will be detailed next.

4.2 Virtual Beamforming: Match Path Angles

E-Mi's virtual beamforming (vBeam) algorithm serves two purposes: *First*, the AoAs and AoDs identified above do not have a pairwise mapping. The vBeam can pair up the AoA and AoD values that belong to the same path. *Second*, the received signals $\mathbf{S}$ are a mix from all propagation paths. To estimate the length of each path (Sec. 4.3), their signals have to be separated from each other.

The basic idea is to process the received signal matrix $\mathbf{S}$ offline and emulate Tx/Rx beamforming towards specific directions. This allows us to generate arbitrary beam patterns, bypassing the codebook constraint of phased arrays. Then, vBeam uses a *beam matching metric* to single out each pair of AoD and AoA directions that belong to the same path.

Beam generation: vBeam generates weight vectors of specific beam patterns and applies them to signals from different antenna elements. Whereas the weight vectors can be computed using conventional delay-sum beamformer [38], vBeam applies a beam-nulling technique instead, which beamforms to the desired AoA/AoD directions while nulling signals from other AoA/AoD directions. *This effectively steers the phased-array's sidelobes toward directions from which there is no signal coming, and thus helps suppress irrelevant signals.*

Suppose the AoA and AoD identified above are denoted by vectors $\mathbf{\Theta} = [\theta_1, \theta_2, \ldots, \theta_K]$ and $\mathbf{\Phi} = [\phi_1, \phi_2, \ldots, \phi_K]$, with K being the number of dominant paths. Denote $\mathbf{a_r}(\theta_i)$ and $\mathbf{a_t}(\phi_i)$ as column weight vectors that beamform toward AoA/AoD angle θ_i and ϕ_i. Take the Rx-side as an example, the nulling beam vector $\mathbf{a}_\mathbf{r}^{null}(\theta_i)$, which beamforms to θ_j for $j = i$ and nulls other θ_j for $j \neq i$, can be directly derived from $\mathbf{a_r}(\theta_i)$ by [39].

Beam matching: Suppose vBeam beamforms towards AoA angle θ_i and AoD angle ϕ_j using the foregoing approach. In order to determine whether θ_i and ϕ_j belong to the same propagation path, we design a *beam matching metric* $\mathbf{F}$, which manifests a high value if and only if θ_i and ϕ_j match to the same path. $\mathbf{F}$ is computed by:

$$\mathbf{F}[i, j] = \mathbb{E}[|\mathbf{a}_\mathbf{r}^{null}(\theta_i)\mathbf{S}\mathbf{a}_\mathbf{t}^{null}(\phi_j)^H|^2],\ \forall\ 1 \leq i,\ j \leq K,$$

Algorithm 1 Virtual Beamforming

procedure *vBeam*($\mathbf{\Theta}$, $\mathbf{\Phi}$, $\mathbf{S}$)
  for i = 1:K, j = 1:K **do** ▷ Loop for Rx and Tx arrays
    $F[i,j] = \mathrm{E}[|\mathbf{a_r^{null}}(\hat{\theta}_i)\mathbf{S}\mathbf{a_t^{null}}(\hat{\phi}_j)^H|^2]$ ▷ Beamform RSS
  end for
  for i = 1:K **do** ▷ Beam matching
    [*I_row, I_col*] ← *max_subscript*($\mathbf{F}$) ▷ Subscript of maximum
    $\mathbf{F}[I\_row,:] \leftarrow 0$; $\mathbf{F}[:,I\_col] \leftarrow 0$;
    $\hat{\mathbf{\Theta}}[i] \leftarrow \mathbf{\Theta}[I\_row]$; $\hat{\mathbf{\Phi}}[i] \leftarrow \mathbf{\Phi}[I\_col]$
  end for
  return $\hat{\mathbf{\Theta}}$ and $\hat{\mathbf{\Phi}}$ ▷ pairwise AoA and AoD
end procedure

where the inner part of above equation applies the virtual beams to signal matrix $\mathbf{S}$, and outer expectation computes the corresponding RSS. Since it is difficult to find an absolute gauge threshold, E-Mi adopts an iterative algorithm (Algorithm 1). It starts with the largest metric and takes the corresponding AoA/AoD as a pair. Then it removes values of pairwise AoA/AoD from the row and column of the matching matrix $\mathbf{F}$ and repeats above procedure to find the next largest matching metric. This approach works well for paths of different signal strengths.

Once vBeam identifies all the pairwise AoA/AoD, it can *isolate path i's signal* $\mathbf{S_i^{path}}$ by projecting the entire signal matrix $\mathbf{S}$ towards path i's AoA and AoD:

$$\mathbf{S_i^{path}} = \mathbf{a_r^{null}}(\hat{\mathbf{\Theta}}[i])\mathbf{S}\mathbf{a_t^{null}}(\hat{\mathbf{\Phi}}[i])^H, \quad (1)$$

where $\hat{\mathbf{\Theta}}$ and $\hat{\mathbf{\Phi}}$ are matrices of the pairwise AoA and AoD. E-Mi then further estimates the RSS of signal isolated from each path.

4.3 Multi-Tone Ranging: Estimate Path Length

E-Mi estimates each path i's length by processing its signals $\mathbf{S_i^{path}}$, using a *multi-tone ranging* mechanism. Modern communication systems such as 60 GHz 802.11ad commonly adopt OFDM, which modulates signals across different frequency tones (called *subcarriers*). The phase of each subcarrier can be measured using built-in channel estimators. Suppose a subcarrier has frequency f_1, then its phase increases linearly with propagation path length d, following $2\pi f_1 d/c$. Our multi-tone ranging leverages the *phase divergence* among OFDM subcarriers, caused by their frequency difference. Given two subcarriers with frequency f_1 and f_2, their phase divergence at distance d equals $\Delta\varphi = 2\pi(f_2 - f_1)d/c$, where c is light speed. f_1, f_2 are known and $\Delta\varphi$ can be measured. So we can easily map $\Delta\varphi$ back to d.

To improve resilience to channel noise, E-Mi harnesses diversity from many subcarriers in 802.11ad-like communication systems. Suppose we have isolated the preamble signal $\mathbf{S_i^{path}}$ along path i (Sec. 4.2). Suppose L subcarriers are located at frequencies $f_1, f_2, \cdots, f_L$, and the Rx-measured phase values are $\varphi_1, \varphi_2, \ldots, \varphi_L$. Then we estimate path i's length via the following optimization framework:

$$d^* = \arg\max_d |\textstyle\sum_{i=1}^{L} e^{j(\varphi_i - \frac{2\pi f_i d}{c})}|. \quad (2)$$

The RHS of Eq. (2) computes the difference between measured phase and theoretical phase over distance in the phaser domain. Essentially, the optimal distance estimation d^* leads to the closest match between these two sets of phase values.

In practice, due to the carrier frequency offset (CFO) between Tx and Rx, the subcarriers will bear unknown phase shifts, which contaminate the phase measurement. We cannot apply the standard CFO compensation technique [40] in this case because it will simultaneously nullify the phase divergence. We address this problem using a *reference calibration* scheme. Specifically, we first separate the Tx and Rx by a known distance d_0 and measure the phase value $\varphi_i(d_0)$ of each subcarrier i. When the Tx/Rx moves to a new (unknown) distance d, the CFO can be canceled by computing their phase difference: $\varphi_i(d) - \varphi_i(d_0) = \frac{2\pi f(d-d_0)}{c}$. Substituting φ_i by $\varphi_i - \varphi_i(d_0)$, Eq. (2) can be reformulated as:

$$d^* = \arg\max_d |\textstyle\sum_{i=1}^{L} e^{j(\varphi_i - \varphi_i(d_0) - \frac{2\pi f_i (d-d_0)}{c})}|. \quad (3)$$

We note that the phase divergence has an aliasing effect: if the phase difference between two subcarriers exceeds 2π, it will wrap and cause ambiguity. To maximize the unambiguous ranging distance, we should maximize the cycle length of phase divergence, or equivalently minimize the frequency separation between subcarriers (denoted as f_{min}). The unambiguous range is thus determined by $\frac{c}{f_{min}}$. For 802.11ad, f_{min} equals the separation between adjacent subcarriers, *i.e.*, 5.156 MHz [2], or equivalent to up to 58.18 m unambiguous ranging distance, which is sufficient for indoor scenarios.

5. Dominant Reflector Reconstruction

The ⟨*AoA, AoD, length*⟩ of all propagation paths form a set of spatial constraints, allowing E-Mi to locate the *reflecting points*, *i.e.*, points where dominant reflectors "bend" the propagation paths. Consequently, E-Mi can geometrically reconstruct reflectors' position, orientation, and reflectivity.

5.1 Locating Reflecting Points in Environment

To locate the reflecting points, E-Mi first *pinpoints the Rx* relative to the Tx, based on the ⟨*AoA, AoD, length*⟩ constraints. Fig. 4 shows an example. Suppose we obtained the ⟨*AoA, AoD, length*⟩ of a single path. Then any point along *line segment* AB satisfies the same ⟨*AoA, AoD, length*⟩ constraint, and is likely to be the Rx position. Therefore, a single path cannot pinpoint the Rx. But we can resolve the ambiguity by adding another path: the intersection between line segment AB of one path and segment $A'B'$ of another path pinpoints the Rx location.

Practical environment may encounter more than two paths. Denote path i's ⟨*AoA, AoD, length*⟩ as $\hat{\theta}_i$, $\hat{\phi}_i$ and

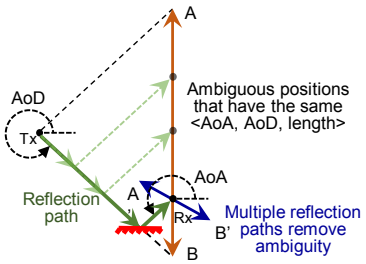


Figure 4: Using a single Tx to locate the Rx and reflecting points. Solid green line represents one propagation path. Dotted green lines are alternative paths with the same $\langle$*AoA, AoD, length*$\rangle$ constraint.

$\hat{d}_i$. Following Fig. 4, we can use a simple geometry to represent the intersection formed by all line segments:

$$\mu_i x + \nu_i y = \gamma_i, \tag{4}$$

$$\begin{aligned}\text{with } \mu_i =& \sin(\hat{\theta}_i) + \sin(\hat{\phi}_i),\ \ \nu_i = -(\cos(\hat{\theta}_i) + \cos(\hat{\phi}_i)) \\ \gamma_i =& x_{tx}(\sin(\hat{\theta}_i) + \sin(\hat{\phi}_i)) - y_{tx}(\cos(\hat{\theta}_i) + \cos(\hat{\phi}_i)) \\ &+ \hat{d}_i(\sin(\hat{\phi}_i)\cos(\hat{\theta}_i) - \cos(\hat{\phi}_i)\sin(\hat{\theta}_i))\end{aligned}$$

where (x_{tx}, y_{tx}) and (x_{rx}, y_{rx}) are the Tx and Rx position, respectively. Assuming the Rx position is intersected by N line segments, Eq. (4) can be rewritten in a matrix format:

$$\Gamma X = P, \tag{5}$$

where $X = [x_{rx}, y_{rx}]^T$, $\Gamma = [\mu_1, \ldots, \mu_K; \nu_1, \ldots, \nu_K]^T$, and $P = [\gamma_1, \ldots, \gamma_K]^T$. In practice, due to residual error of measurement, line segments may not intersect on a single point. We thus reformulate Eq. (5) as a least-square optimization problem:

$$X^* = \arg\min_X ||P - \Gamma X||^2, \tag{6}$$

where X^* estimates the Rx position with minimum error.

Unfortunately, our initial experimental tests found the optimization alone works poorly due two practical factors: *(i)* The $\langle$*AoA, AoD, length*$\rangle$ estimation (Sec. 4) contains residual errors, especially for long-range and weak-RSS paths. Such errors may cause intersecting line segments to be close to parallel, which significantly amplifies the Rx location error. *(ii)* The MRF (Sec. 4) may capture high-order reflections that do not follow the model in Fig. 4. Such mismatch may deviate the estimation arbitrarily away from the real position.

E-Mi introduces two mechanisms to overcome above challenges.

(1) Weighting the residual error. We first reformulate the optimization problem in Eq. (6) to account for MRF's residual errors. The inner term of Eq. (6) calculates the difference between matrices ΓX and P. Since different paths' geometries have different residual errors, we weight the paths according to the *confidence level* in MRF's estimation, based on the following observation: those paths of shorter length (thus higher RSS) tend to have less error. Thus, we can use the inverse of path length as the weight. Besides, we need to minimize the sum of distance from Rx position to all line segments, which requires normalization by a coefficient $\sqrt{\mu_i^2 + \nu_i^2}$ [41]. The final weight value for path i is:

$$w_i = \frac{1}{d_i\sqrt{\mu_i^2 + \nu_i^2}},$$

which consists of the inverse of path length and the normalization factor. Stacking the weights into a vector $W = diag(w_1, \ldots, w_K)$, the optimization problem Eq. (6) can be rewritten as $X^* = \arg\min_X ||W(P - \Gamma X)||^2$, which can be solved by standard least-square algorithms.

(2) Filtering higher-order reflection. To constrain the problem within the geometrical model of first-order reflection (Fig. 4), we should exclude any higher order reflection paths in the optimization. Our key observation is that line segment intersections of higher-order reflections tend to randomly distribute and exhibit a larger deviation since they do not fit into the geometry model for the first-order reflection. Therefore, we apply a K-means clustering algorithm to filter out the $p\%$ most significantly deviated line segments that most likely belong to the higher-order reflection. The choice of p value depends on the amount of higher-order reflections. We prefer a larger p value in a highly reflective environment, and otherwise a smaller p. Yet, we find E-Mi is not sensitive to it because most indoor environments have comparable number of dominant reflectors, and an empirical value (e.g., $p = 20$) would suffice.

After determining the client position, the *locations of reflecting points* (x^i_{ref}, y^i_{ref}) can be estimated by:

$$\frac{x^i_{ref} - x_r}{y^i_{ref} - y_r} = \tan(\hat{\theta}_i),\ \ \frac{x^i_{ref} - x_t}{y^i_{ref} - y_t} = \tan(\hat{\phi}_i), \tag{7}$$

which solves a set of equations following simple geometry in Fig. 4.

5.2 Reconstructing Dominant Reflector Layout and Reflectivity

E-Mi reconstructs the dominant reflector geometry (orientation/location/length) and reflectivity, by sampling the wireless channel across a *sparse* set of Rx locations, and locating the corresponding reflecting points following the above steps. It creates a 2-D cross section of the environment corresponding to the horizontal plane of the Tx/Rx. Extension to the 3-D case will be discussed in Sec. 10.

Reconstructing dominant reflectors' geometry. Ideally, we can move the Rx to many positions, each helping to locate multiple reflecting points. A sufficient number of reflecting points can form a *pixel cloud* that outlines the reflector geometry. However, due to sparsity of the propagation paths [9–11], collecting a dense pixel cloud requires hundreds of Rx positions even for a small office.

To avoid such war-driving, we design a sparse reconstruction method which only samples at a few positions. We abstract the reflectors into two categories. *Specular*

Reflector Reflector

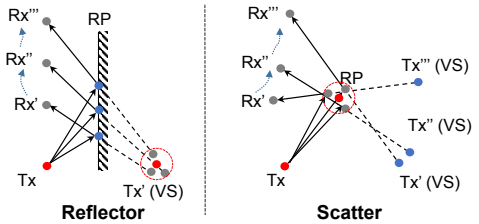


Figure 5: Reflecting point (VP) and corresponding virtual source (VS), used to recover the geometry and location of specular/diffusive reflectors.

reflectors: a large continuous surface, *e.g.*, wall and cabinet, upon which mmWave signals experience specular reflection [7]; *Diffusive reflectors:* small-size objects, *e.g.*, pillar and computer screen, which scatter signals towards a wide range of angles [42].

(i) Locating specular reflectors. We model a specular reflector as a continuum of planar segments, each with different reflectivity or orientation. According to the law of reflection, signals that are reflected by a specular reflector look like emitting from a virtual source (Fig. 5). Given the ⟨*AoA, AoD, length*⟩ and reflecting point location of one propagation path, we can use simple geometry to pinpoint the virtual source (VS) position relative to the real TX position, following the law of reflection.

Ideally, each specular segment should have a single VS even as the Rx moves and creates different reflection paths. Yet due to the MRF's residual errors, the VSs estimated at different Rx positions do not exactly overlap, but fortunately they tend to form a cluster (Fig. 5). We apply the clustering algorithm [43] to isolate the clusters and use each cluster center to represent one VS. Once a reflector segment's VS is identified, its position and orientation is readily determined via a mirror-partitioning between the real source and VS. Further, we extend the reflector segment, and take the size of geometrical shape from pixel cloud as constraint to determine the length of each reflector segment. An example experiment will be provided in Sec. 8.2 to elucidate the procedure.

(ii) Locating diffusive reflectors. Observing that a diffusive reflector corresponds to densely concentrated reflecting points (Fig. 5), we can also apply the clustering algorithm on the reflecting points and use the centers of resulting clusters to represent the diffusive reflectors. Yet this will be interfered by dispersive reflecting points created by specular reflectors. Fortunately, based on the previous step, we can identify and exclude such interfering points. Specifically, we identify them based on their Euclidean distance to the specular reflectors. Threshold is set to $3\times$ the variance of reflector position error, which can isolate a majority of specular reflecting points.

Estimating reflection loss. We now describe how E-Mi models reflection loss, the major distorting factor when signals hit the reflector. Other factors such as diffraction may also vary the signal strength but the effect is minor [44]. E-Mi separately models the reflection loss of each path it has identified. Three factors contribute to the propagation loss: free-space pathloss, oxygen absorption O_l and reflection loss R_l, *i.e.*,

$$RSS = P_t + G_t + G_r - \big(20\log_{10}(d) + O_l + R_l\big),$$

where d is the path length. P_t, G_t, G_r represent the Tx power and Tx/Rx antenna gain. The O_l almost remains a constant for distance of tens of meters [36]. For each path, d and *RSS* are known from MRF (Sec. 4). Thus, to obtain R_l, we need to obtain the constant parameters P_t, G_t, G_r, which may not be available in practice. In addition, the constant value O_l is unknown either.

We address this issue by using the LOS path as reference calibration to cancel those unknown factors. First, we can isolate the LOS path's signals from NLOS paths' signals by metrics such as shortest path length and strongest RSS. We then estimate the reflection loss of each NLOS reflecting path via a simple subtraction:

$$R_l = RSS_{LOS} - RSS_{ref} - 20\log_{10}(d_{ref}/d_{LOS}), \quad (8)$$

where RSS_{LOS}, RSS_{ref} and d_{LOS}, d_{ref} are RSS and path length for LOS and reflected path. Since each reflecting segment/point may have multiple estimations corresponding to multiple reflection paths, we take the average as its final reflection loss.

6. Parametric Ray-tracing: Predict Link Performance

Ray-tracing is a fine-grained way to model wireless signal propagation in both indoor and outdoor environment [45, 46]. It tracks the details of how *each signal path* is attenuated over distance and reshaped by reflectors. Measurement studies demonstrated that, given a precise physical description of reflectors, the signal pattern predicted by ray-tracing is reasonably close to real measurement in both LOS and NLOS scenarios [47].

E-Mi employs a parametric ray-tracing engine, whose input is the aforementioned layout/reflectivity for dominant reflectors constructed directly from the 60 GHz radios' eyes. We develop E-Mi's ray-tracer following the classical approaches in [48, 49], which models the signal propagation in a 2-D domain using a geometrical/optical tracing. The ray-tracer captures the attenuation and reflection effects along all paths, and recursively traces a path until it attenuates by more than 30 dB. In addition, the ray-tracer accounts for the angle-dependent antenna gain patterns from phased-arrays. The gain patterns can be obtained from either phased-array simulators or hardware specification. After synthesizing signals from all paths, the ray-tracer outputs the final RSS and converts it to bit-rate following a standard 802.11ad rate table [11].

7. Implementation

We prototype E-Mi on a custom-built 60 GHz software-radio testbed, which uses WARP [50] to generate and

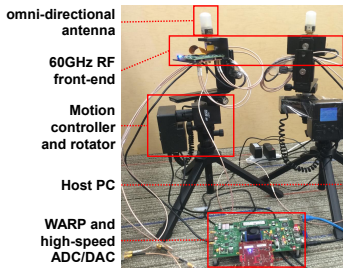


Figure 6: Left: Custom-built 60 GHz software-radio platform. Top-right: Antenna placement of a 16-element phased-array. Bottom-right: Example beam pattern.

process baseband waveforms under the control of a host PC. The digital signals are converted to/from analog through a high-speed DAC/ADC, and carrier-modulated by the Pasternack PEM-003 60 GHz RF front-end [51].

The PHY layer metrics (*e.g.*, channel state information) in commercial 60 GHz radios, though internally available to the radio vendors, are still not opened to the public yet. The recently developed phased-arrays in 60 GHz software radios [52] can only support a small number of elements. Therefore, we reproduce the effects of a 60 GHz circular phased array using the time-lapse synthesis method, which follows the way that a phased array modulates, weights and transmits wireless signals. We mount a 60 GHz omni-directional antenna MI-WAVE 267V (with 360° azimuth and 40° elevation beamwidth) [53] onto Axis360 [54], a programmable motion control system (Fig. 6). The Axis360 rotates the antenna to discrete positions along a circle, each position corresponding to one antenna element on a real circular array. The array dimension follows empirical recommendations in antenna design [39]. The radius of a 16-element circular array is 6.4 mm, with 2.5 mm (half-wavelength) between adjacent elements. This time-lapse synthesis approach has been adopted and verified by previous works [55,56].

To synthesize a pair of Tx/Rx phased-arrays, we apply beamforming weights (based on a standard 802.11ad codebook [35]) and then combine the measurement from all elements within the Rx circular array. Since E-Mi runs in static environment, this time-lapse approach can realistically reproduce a real phased-array where all elements are excited concurrently. Besides, each element of a phased array antenna is expected to have a close to omnidirectional coverage in horizon plane [57]. Fig. 6 depicts an example Tx phased-array gain pattern generated by this time-lapse approach, and measured using a 3.4° horn receiver. Despite the measurement speed of our platform is slow currently owing to the mechanical antenna rotation, a full-fledged 60 GHz device, that has the electronic phased array antenna, can transmit a wireless packet at tens of microseconds. The overall sensing time of each location will be at millisecond-level.

We implement E-Mi's major modules (Sec. 3) within

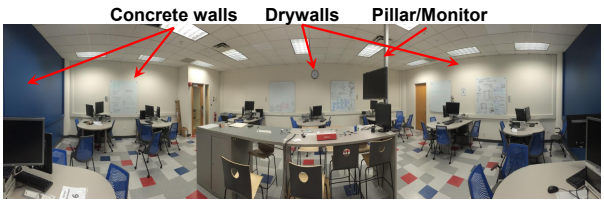


Figure 7: Dominant reflectors in an office environment.

the software-radio's host PC. Due to limited bandwidth, our platform cannot send 802.11ad-compatible preambles for channel estimation. Instead, the Tx sends five orthogonal tones from 3 MHz to 15 MHz as baseband signals, modulated by 60.48 GHz carrier frequency. This does not obstruct our validation because the narrow band implementation can be considered as only utilizing a few subcarriers in the 2 GHz wide band. When an 802.11ad-compatible device is available, E-Mi can be easily extended to conduct MRF and dominant reflector reconstruction across orthogonal subcarriers over a wideband.

For evaluation purpose, we also use the beam-scanning method to acquire the ground-truth ⟨*AoA, AoD, length*⟩, similar to Rappaport *et al.* [9]. We use a pair of Tx/Rx radios equipped with directional horn antennas of 3.4° beamwidth [58]. With Axis360, the Tx antenna sweeps the horizontal plane at a step of 3°. Meanwhile, the Rx measures the wireless channel and steers to next step after the Tx completes a full scanning.

8. Experimental Validation

8.1 Effectiveness of Multipath Resolution

To verify the MRF (Sec. 4), we set up a pair of Tx and Rx, each synthesizing a 16-element phased-array. We conduct experiments in a 90 m^2 office environment, which represents a typical indoor environment. The dominating reflectors involve 2 drywalls, 2 concrete walls and 1 pillar (Fig. 7). We fix the Tx and randomly move Rx over multiple locations. The result is compared against the ground-truth AoA, AoD (measured using the oracle beam scanning, Sec. 7) and path length (measured using a laser ranger). The ground-truth measurement reveals each link has 3 to 5 dominating propagation paths.

Success rate of AoA/AoD detection: Recall that MRF needs to *detect* and then *pair* each AoA/AoD that belongs to the same path (Sec. 4). Our measurement shows that MRF correctly detects 89% and 82% of AoAs and AoDs, and almost 100% of the correctly detected AoAs/AoDs are correctly paired. MRF fails to detect AoA/AoD of some paths primarily because their reflected signal strength is too weak – We find the RSS of unidentified paths is typically 16 dB lower than the LOS path. In other words, the detection failure is not critical since they will have a limited impact on link performance.

Accuracy in resolving AoA/AoD: Fig. 8 depicts how

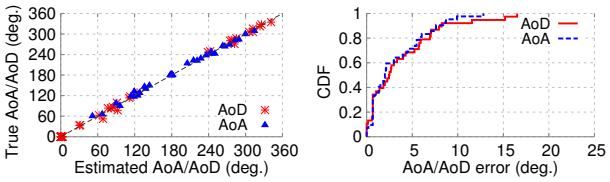


Figure 8: Left: Estimated AoA/AoD from MRF v.s. ground truth; Right: CDF of AoA/AoD error from MRF.

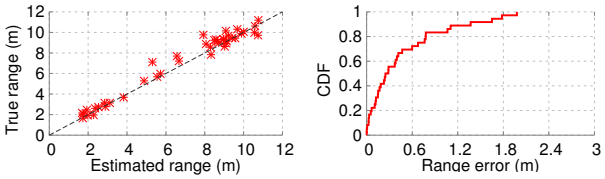


Figure 9: Left: Estimated path length v.s. ground truth; Right: CDF of path length error from MRF.

accurately E-Mi can resolve the AoA/AoD of all the multipaths that it successfully detects. We observe that *the estimated direction closely matches the true direction*. On average, the AoA/AoD error is only 3.5° and the 90-th error is 8°. Considering that the beam-switching granularity of a 16-element phased-array is 22.5° [35], the *accuracy of AoA/AoD estimation indeed bypasses the granularity constraint of codebook-based beamforming and achieves super-angular resolution*. We expect the accuracy can be improved by larger phased-arrays, due to more entries in the received signal matrix (Sec. 4).

For comparison, we also run the codebook-based beam-scanning method (Sec. 4). We found its success rates in detecting AoA, AoD and pairing the AoA/AoD is only 64.4%, 66.7% and 53.3% respectively. And the average estimation error of AoA and AoD are 21.0° and 22.1°, respectively. The fundamental reason lies in the aforementioned sidelobe problem (Sec. 4). *This experiment further verifies the necessity and effectiveness of the virtual beamforming method in MRF, which pairs up AoA with AoD while nullifying sidelobes*.

Accuracy of path length estimation: We run E-Mi's multi-tone ranging mechanism over all detected paths. The scatter plot in Fig. 9 shows the estimated length v.s. true length. E-Mi achieves an average error of only 0.4 m and 90-th error of 1 m. The LOS paths (typically < 4 m) tend to have smaller estimation error (0.23 m on average) than NLOS reflection paths due to better RSS. E-Mi can achieve this ranging accuracy using even a relative small bandwidth because the vBeam algorithm (Sec. 4.2) can isolate the signals from different paths. This accuracy is sufficient for most 60 GHz applications since the prediction metrics (AoA/AoD and signal strength) are not very sensitive to the range measurement error (by the Friis law, 1 m error only causes less than 2 dB path loss deviation). It is expected that the ranging error will further reduce (Sec. 4.3) in practical 802.11ad radios with 1.7 GHz bandwidth. To summarize, this microbenchmark verifies

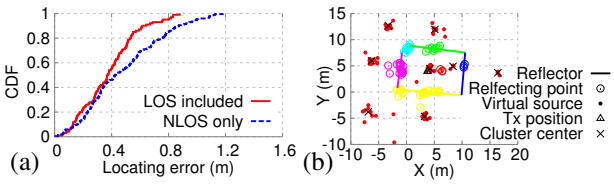


Figure 10: (a) Error CDF in the reflecting point localization algorithm. (b) Breakdown of reconstruction output for an office environment.

Reflector color	Reflection loss	Reflector color	Reflection loss
drywall	15.87 dB	concrete wall	8.60 dB
drywall	13.83 dB	drywall	20.56 dB
concrete wall	5.50 dB	pillar/monitor	1.24 dB

Table 1: Estimated reflection loss on different reflectors.

that *the multi-tone ranging precision in E-Mi is sufficient to support dominant reflector reconstruction, even when using our low-end communication hardware*.

8.2 Effectiveness of Dominant Reflector Reconstruction

Following the setup in Sec. 8.1, a static Tx executes the algorithm in Sec. 5.1 to locate the reflecting points for each given Rx position. We move the Rx to 15 uniform positions to reconstruct the dominating reflectors.

Accuracy in localizing reflecting points. Since the reflecting point location has a linear, deterministic relation with the Rx position (Sec. 5.1), we mainly focus on evaluating the latter, whose ground truth is obtained via a laser range finder BOSCH DLE40.

Fig. 10 (a) plots the CDF of localization error. E-Mi can locate the Rx position with mean/90-th error of 0.38 m and 0.6 m, which is even smaller than the path length estimation, because we apply the minimum least square method that leverages the redundant information of multiple reflected paths to reduce the estimation error. Indeed, when we intentionally eliminate the LOS paths, the performance ("NLOS only") drops due to lower path diversity. The results verify that E-Mi's *reflecting point localization algorithm indeed achieves high precision based on the MRF. More paths provide more diversity and hence higher accuracy*.

Performance of dominant reflector reconstruction. Recall that, given an estimation of the Rx's and reflecting points' positions, the dominant reflector reconstruction locates the virtual source, specular reflector, and diffusive reflector, respectively. Fig. 10 (b) puts together the output from each step, and shows the final reconstruction output based on 15 Rx sampling positions. We observe that the output closely matches the ground truth: 4 walls (specular reflectors) and one pillar/monitor (diffusive reflector). In effect, even the geometrical size of the reflectors matches the ground truth well with less than 0.3 m error. Table 1 lists the estimated reflection loss. The two concrete walls have $10+$ dB lower loss than the two drywalls. And the metal pillar/monitor shows even lower

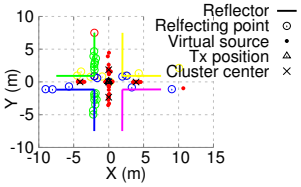


Figure 11: Breakdown of reconstruction output for the corridor environment.

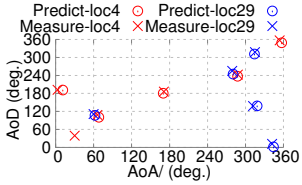
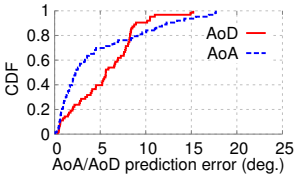


Figure 12: Predicted v.s. measured AoA/AoD patterns for two locations.



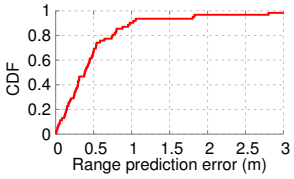


Figure 13: Left: Error in predicting the AoA/AoD of unobserved locations. Right: Error in predicting the path length of unobserved locations.

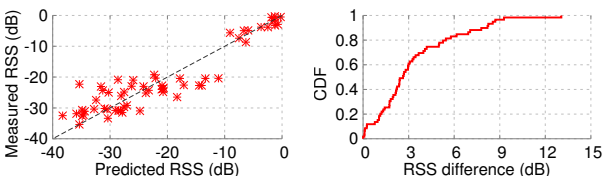


Figure 14: Left: Predicted RSS v.s. measured RSS; Right: CDF of RSS difference between prediction and measurement.

Figure 15: Experiment in a complicated printing room.

loss. Intuitively, this matches with the reflectivity of materials at 60 GHz [59]. We have also evaluated E-Mi in the cross of a corridor (Fig. 11) and observed a similar level of accuracy in positioning the dominant reflectors.

In summary, *E-Mi can effectively identify the geometry, locations and distinguish the reflectivity of major reflectors from the 60 GHz radio's eyes*.

Accuracy of link performance prediction. We now feed the reconstructed reflector geometry/reflectivity into the parametric ray-tracing engine, and predict the spatial channel for another set of 15 randomly located Tx/Rx pairs. Following the ground-truth measurement (Sec. 7), we found these links have 66 paths in total. Fig. 12 showcases example results from two randomly selected links. We observe that the predicted AoA/AoD patterns are highly consistent with the ground truth. The missing spot (*e.g.*, AoA 30°/AoD 40°) is caused by reflection that is not captured by the Tx/Rx during MRF. We found such spots correspond to signal paths with negligible RSS and hence little impact on network performance. Also, adding more Rx position samples can incrementally reduce the probability of prediction loss.

Fig. 13 plots the channel prediction error over all propagation paths among all links. The average and 90-th errors of path length are 0.64 m and 1.41 m. The average AoA and AoD prediction errors are $4.5°$ and $5.7°$, and 90-th errors are $12.2°$ and $10.0°$, respectively. These results verify that *E-Mi can accurately predict the AoA/AoD and path length of unobserved locations, based on a number of sparse samples.*

Fig. 14 further shows the predicted v.s. measured RSS among all paths and the corresponding CDF. We observe that *among all paths and links, E-Mi's median RSS prediction error is only 2.8 dB*. The scatter plot in Fig. 14 further shows that locations with higher RSS benefits from higher prediction accuracy, since it mainly involves LOS paths that follow the Friis pathloss model more closely. Even though E-Mi's prediction is imperfect, we show that it can already become a salient tool for network deployment and protocol optimization (Sec. 9).

Scalability in complicated environment. We explore the scalability and generality of E-Mi in two steps. *First*, we collect ground-truth channel profile in a printing room – a more complicated environment (Fig. 15) than the office. Fig. 16 (a) and (b) compare the channel profiles (high RSS corresponding to the AoAs/AoDs created by dominant reflectors). Although the printing room hosts a much larger number of objects, the number of dominant reflectors remains similar (~ 6). Our close examination reveals that the dominant directions mainly come from the metal shelf, glass window and concrete wall, *etc.*, *i.e.*, large objects with strong reflectivity. On the other hand, wooden shelf, desk, and small structures on the wall, barely contribute to the RSS. In both environments, the 6 most dominant reflections account for $> 95\%$ of the total RSS. This experiment demonstrates that *even in a complicated environment, the 60 GHz channel remains sparse, i.e., only a few dominant reflectors determine the channel quality.* Thus, to predict the channel, E-Mi only needs to capture and model a few dominant reflectors.

Second, we investigate how many sampling locations are needed to reconstruct the dominant reflectors. We simulate an 8×10 m^2 room environment with four concrete walls. The Tx is placed at the center and Rx randomly moves over $5 \sim 30$ locations in each test. Across tests, different number of planar reflectors are placed randomly around the Tx/Rx locations. We use the ray tracing to obtain the ground-truth $\langle$*AoA, AoD, length*$\rangle$. Fig. 17 (a) depicts the average error between E-Mi's reconstructed reflector positions and ground truth. Generally, *more dominant reflectors requires more sampling positions*. Yet, even for an 8-reflector environment, E-Mi only needs 20 sampling locations to ensure an accuracy of around 0.2 m. Since the number of dominating reflectors tends to be

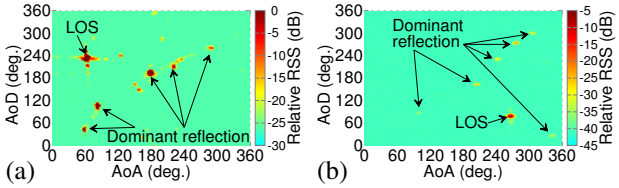


Figure 16: Spatial channel profile of (a) a printing room with many reflectors and (b) an office environment.

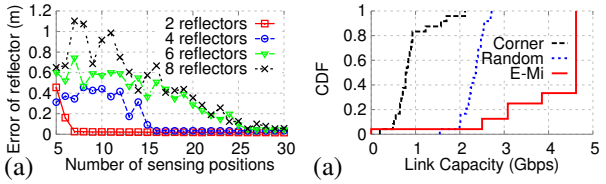


Figure 17: (a) Reflector location error vs. number of Rx sampling positions. (b) Client throughput CDF under a 2-AP architecture.

sparse, the amount of on-site sampling needed in E-Mi is still substantially lower than that of a site-survey. Besides, the reflector accuracy depends on the radio's sensing position, and it tends to be more accurate if the radio can see stronger reflected paths from a reflector. Fortunately, this location sensitivity will be averaged out when the number of sensing positions is sufficiently large.

9. Case Study of E-Mi

In this section, we present an example application which uses E-Mi to predict 60 GHz network performance and optimize the AP placement. E-Mi can also be applied to other scenarios, *e.g.*, device localization and environment mapping. Yet, exploring E-Mi in a broader range of areas is beyond the scope of this work.

9.1 Environment-Aware 60 GHz AP Deployment

Measurements (Sec. 1 and [11]) showed that *the performance of 60 GHz networks is a sensitive function of location and reflector position*, specifically *w.r.t.*

(i) Coverage: The spatial RSS distribution of a 60 GHz AP tends to be unevenly distributed, even among same-distance links, due to two unique factors: *(a)* High directionality: The AP's phased-array antenna can only generate a discrete set of directional beam patterns, which typically point to unevenly distributed spatial angles [11, 36]. *(b)* Ambient reflections: Different reflectors cause RSS to distribute unevenly, *e.g.*, receivers close to strong reflectors tend to benefit from high RSS [11].

(ii) Robustness: i.e., resilience of the network under blockage. 60 GHz links tend to be frequently disrupted due to inability to diffract/penetrate human body. Beam-steering alleviates the problem, but whether the resulting reflection path can detour blockage highly depends on the geometry/reflectivity of environment [11, 60].

To address the environment sensitivity, we propose to use a multi-AP architecture to cover a constrained environment. Through a central controller, the APs can

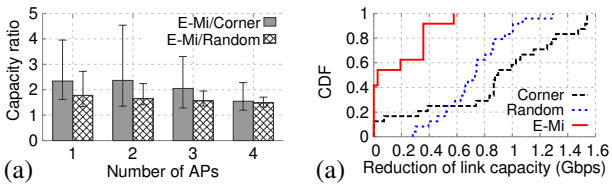


Figure 18: (a) Coverage improvement over alternative strategies. (b) Throughput loss under random blockage.

tightly cooperate with each other. When experiencing poor throughput or blockage, a client can immediately switch to an alternative AP. The architecture itself is not new, but in 60 GHz environment, it needs to meet one key challenge: *For a set of APs under a given environment, how to deploy them optimally so as to maximize the coverage and robustness to blockage?*

We employ E-Mi to answer this question. The basic idea is to predict the AP locations that provide best coverage and robustness to a typical set of client spots. The *client spots* are locations at which clients tend to concentrate. The client spots can be manually specified by users within E-Mi's environment map (*e.g.*, Fig. 10 (b)). Alternatively, the E-Mi AP can divide the environment map into grids, implicitly sense clients' locations over time (Sec. 5.2), and incrementally build clients' spatial distribution. Client spots can be defined as grids where the dwelling probability exceeds a threshold (*e.g.*, 0.05).

To maximize coverage, we define the performance metric $\mathcal{D}$ as the *mean bitrate* among W given client spots. Suppose each client is associated to the AP with highest bitrate. Then, for each combination of AP locations,

$$\mathcal{D} = \frac{1}{W}\sum_{i=1}^{W} \arg\max_{\{j=1,\cdots,A\}} T(RSS)_{ij}, \qquad (9)$$

where RSS_{ij} represents the RSS from AP j to client i. A is the number of APs, and $T(\cdot)$ maps RSS to the achievable bitrate using the 802.11ad rate table [32]. The candidate AP position can either fall in the grids or be manually specified by users within E-Mi's environment map. To find the best multi-AP deployment, we simply use E-Mi to predict the performance of all candidate position combinations and select the one that maximizes $\mathcal{D}$.

For a given multi-AP deployment, we further define the *robustness metric* as $\mathcal{E} = \mathcal{D}'/\mathcal{D}$, where $\mathcal{D}'$ is the mean bitrate under blockage. Using E-Mi, we can "rehearse" the impact of human body blockage without field war-driving. Since human body is aquaphobic [11, 36, 61], the blockage may annihilate one or more paths. So we can use ray-tracing to derive $\mathcal{D}'$ by averaging the bitrate resulting from blockage of random movement. We repeat the procedure over candidate AP position combinations and single out the best. The optimized AP location essentially maximizes the number of paths to each client. The more reachable paths will make the network more resilient to blockage because radios can immediately reestablish the link upon blocking via another path.

9.2 Experimental Verification

We evaluate the E-Mi-based AP deployment with dominant reflectors in Fig. 10 (b). We set 24 random client spots and divide the environment into 20 equal sized grids. The center of each grid is considered as a candidate AP position. We compare E-Mi with *random* deployments and an empirical approach that puts APs in four *corners* to maximize coverage. Since our experiments have already validated the accuracy of the ray tracing method, we reuse ray tracing to evaluate the RSS from an AP to client. The RSS is then mapped to bit-rate and link throughput following [11]. Consistent with Sec. 8.1, each AP/client has a 16-element phased array, with 32 codebook entries and beamwidth of $22.5°$.

Coverage: Fig. 17 (b) plots the CDF of clients' throughput under 2-AP deployment. For *Random*, the throughput is averaged across all AP locations. We observe that E-Mi gains substantial advantage from its environment-awareness, with median throughput improvement of $2.24\times$ over Random, and $4.54\times$ over Corner. Moreover, E-Mi consistently delivers higher capacity for all clients, *i.e.*, it does not sacrifice fairness. The results manifest the ineffectiveness of empirical approaches, which are unaware of the impact of dominant reflectors on 60 GHz network performance. Even with 2 APs, E-Mi can optimize AP placement and boost network capacity to $1.4\times$ compared with a single-AP deployment.

We further evaluate E-Mi in 30 different environment topologies, created by intentionally adding reflectors (up to 10), with random orientations, inside the environment. Fig. 18 (a) shows that, given a single AP, E-Mi has $2.2\times$ average capacity gain over the empirical deployment, and up to $4\times$ gain in certain environment that is observed to feature heterogeneous reflector placement. More APs can offset the environment heterogeneous, and hence degrade E-Mi's gain slightly.

Robustness: Under the same topology as Fig. 17 (b), Fig. 18 (b) plots the CDF of throughput loss across all clients under random human blockages (created by randomly moving at different locations inside the environment). Owing to its awareness of reflectors, E-Mi's median throughput loss is only around 20 Mbps, in comparison to 700 Mbps to 830 Mbps in the empirical approaches. The normalized throughput gain of our optimization is consistent with the measurement (Fig. 1) using commercial 60 GHz devices.

We also found that the coverage-maximizing deployment may differ from the robustness-maximizing one. In practice, one may use a weighted balance between the metrics, based on how likely the blockage is to happen.

10. Discussion and Future Work

Using E-Mi in commodity phased-arrays: Our evaluation used a virtual array of 16 omni-directional antenna elements to synthesize a phased-array. Commodity phased-arrays may have a limited field-of-view angle, and their beams reside within half-space ($180°$) [33]. However, as long as the codebook and gain pattern are available (usually specified by device manufacturers), E-Mi's multipath resolution framework is applicable. In addition, we can flip the phased-arrays' orientation to ensure full-space coverage.

From 2-D to 3-D sensing: Our E-Mi design places the Tx/Rx on the same height and senses a 2-D cross-section. Extending E-Mi to the 3-D case involves some new challenges, *e.g.*, resolving AoA/AoDs along both azimuth and elevation dimensions. However, the main design principles of E-Mi can still apply. Notably, the geometry of dominant reflectors along the vertical dimension (mostly floors and ceilings) is much simpler and easier to estimate. A user can even directly provide the height information to assist E-Mi in estimating the dominant reflectors in 3-D. We leave such exploration to future work.

Sensing complicate-structured environment: E-Mi abstracts the 2-D environment as a composition of lines (for specular reflectors) and spots (for diffusive reflectors). The abstraction is accurate if the radio environment is sparse, *i.e.*, dominated by large reflectors like walls and furnitures (*e.g.*, cabinet, bookshelf, and refrigerator). Environmental sparsity in turn causes channel sparsity, which has been observed by many 60 GHz measurement studies [9, 18, 62, 63]. Nonetheless, E-Mi cannot capture complicated structures. These structure may violate the channel sparsity assumption and exacerbate higher order reflections. E-Mi does not attempt to capture mobile structures either, as clarified in Sec. 3.

11. Conclusion

We present E-Mi as a sensing-assisted paradigm to facilitate 60 GHz networks, whose performance is highly sensitive to reflectors. E-Mi senses the environment from 60 GHz radios' eyes. It "reverse-engineers" the geometry/reflectivity of dominant reflectors, by tracing back the LOS/NLOS paths between a pair of 60 GHz nodes. Through case studies and testbed experiments, we have demonstrated how such environment information can be harnessed to predict 60 GHz network performance, which can in turn augment a broad range of network planning and protocol reconfigurations.

Acknowledgement

The work reported in this paper was supported in part by the NSF under Grant CNS-1343363, CNS-1350039, CNS-150665, and CNS-1518728. It was also partly supported by National Natural Science Foundation of China No. 61572476, No. 61532012, No. 61332005, No. 614-21061 and the Beijing Training Project for the Leading Talents in S&T ljrc201502.

12. References

[1] B. Wire, "FCC Adopts New Rules for Unlicensed V-Band, Extending the Coverage to 57-71 GHz," 2016.

[2] IEEE Standards Association, "IEEE Standards 802.11ad-2012: Enhancements for Very High Throughput in the 60 GHz Band," 2012.

[3] Qualcomm, "Qualcomm 802.11ad Products to Lead the Way for Multi-band Wi-Fi Ecosystem," https://www.qualcomm.com/news/releases/2016/01/05/qualcomm-80211ad-products-lead-way-multi-band-wi-fi-ecosystem, 2016.

[4] Google, "Tango," https://get.google.com/tango/, 2014.

[5] B. Ferris, D. Fox, and N. Lawrence, "WiFi-SLAM Using Gaussian Process Latent Variable Models," in *Proc. of IJCAI*, 2007.

[6] J. Aulinas, Y. Petillot, J. Salvi, and X. Lladó, "The SLAM Problem: A Survey," in *Proc. of the Internation Conference on Artificial Intelligence Research and Development*, 2008.

[7] Y. Zhu, Y. Zhu, B. Y. Zhao, and H. Zheng, "Reusing 60GHz Radios for Mobile Radar Imaging," in *Proc. of ACM MobiCom*, 2015.

[8] R. Appleby and R. Anderton, "Millimeter-Wave and Submillimeter-Wave Imaging for Security and Surveillance," *Proceedings of the IEEE*, 2007.

[9] H. Xu, V. Kukshya, and T. Rappaport, "Spatial and Temporal Characteristics of 60-GHz Indoor Channels," *IEEE Journal on Selected Areas in Communications*, 2002.

[10] P. Smulders, "Exploiting the 60 GHz Band for Local Wireless Multimedia Access: Prospects and Future Directions," *IEEE Communications Magazine*, 2002.

[11] S. Sur, V. Venkateswaran, X. Zhang, and P. Ramanathan, "60 GHz Indoor Networking through Flexible Beams: A Link-Level Profiling," in *Proc. of ACM SIGMETRICS*, 2015.

[12] J. Xiong and K. Jamieson, "ArrayTrack: A Fine-grained Indoor Location System," in *Proc. of USENIX NSDI*, 2013.

[13] K. Joshi, S. Hong, and S. Katti, "PinPoint: Localizing Interfering Radios," in *Proc. of USENIX NSDI*, 2013.

[14] Cisco Inc., "Site Survey Guidelines for WLAN Deployment," 2013. [Online]. Available: http://www.cisco.com/c/en/us/support/docs/wireless/5500-series-wireless-controllers/116057-site-survey-guidelines-wlan-00.html

[15] S. Puthenpura, "Understanding the Science Behind Small Cell Deployment," 2013. [Online]. Available: http://www.research.att.com/articles/featured_stories/2013_11/201311_Small_Cells.html

[16] K.-H. Kim, A. W. Min, and K. G. Shin, "Sybot: An Adaptive and Mobile Spectrum Survey System for WiFi Networks," in *Proc. of ACM MobiCom*, 2010.

[17] J. Robinson, R. Swaminathan, and E. W. Knightly, "Assessment of Urban-scale Wireless Networks with a Small Number of Measurements," in *Proc. of ACM MobiCom*, 2008.

[18] P. F. M. Smulders, "Statistical Characterization of 60-GHz Indoor Radio Channels," *IEEE Transactions on Antennas and Propagation*, 2009.

[19] H. Durrant-Whyte and T. Bailey, "Simultaneous Localization and Mapping: Part I," *IEEE Robotics Automation Magazine*, 2006.

[20] S. Riisgaard and M. R. Blas, "SLAM for Dummies: A Tutorial Approach to Simultaneous Localization and Mapping," 2003, mIT 16-412j Notes.

[21] J. J. Wang, G. Hu, S. Huang, and G. Dissanayake, "3D Landmarks Extraction from a Range Imager Data for SLAM," in *Proc. of Australasian Conference on Robotics and Automation*, 2009.

[22] E. Guizzo, "How Google's Self-Driving Car Works," IEEE Spectrum, 2011.

[23] Z. Yang, C. Wu, and Y. Liu, "Locating in Fingerprint Space: Wireless Indoor Localization with Little Human Intervention," in *Proc. of ACM MobiCom*, 2012.

[24] E. Jose and M. Adams, "An Augmented State SLAM Formulation for Multiple Line-of-Sight Features with Millimetre Wave RADAR," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2005.

[25] F. Guidi, A. Guerra, and D. Dardari, "Millimeter-wave Massive Arrays for Indoor SLAM," in *IEEE International Conference on Communications Workshops (ICC)*, 2014.

[26] R. Schmidt, "Multiple Emitter Location and Signal Parameter Estimation," *IEEE Transactions on Antennas and Propagation*, 1986.

[27] R. Roy and T. Kailath, "ESPRIT-Estimation of Signal Parameters Via Rotational Invariance Techniques," *IEEE Transactions on Acoustics, Speech and Signal Processing*, 1989.

[28] S. S. Hong and S. R. Katti, "DOF: A Local Wireless Information Plane," in *Proc. of ACM SIGCOMM*, 2011.

[29] N. K. Santhapuri, J. Manweiler, S. Sen, X. Bao, R. R. Choudhury, and S. Nelakuditi, "Sensor Assisted Wireless Communication," in *Proc. of IEEE Workshop on Local and Metropolitan Area Networks (LANMAN)*, 2010.

[30] L. Ravindranath, C. Newport, H. Balakrishnan, and S. Madden, "Improving Wireless Network Performance Using Sensor Hints," in *Proc. of USENIX NSDI*, 2011.

[31] T. Nitsche, A. B. Flores, E. W. Knightly, and J. Widmer, "Steering With Eyes Closed: Mm-Wave Beam Steering Without In-Band Measurement," in *Proc. of IEEE INFOCOM*, 2015.

[32] S. Sur, X. Zhang, P. Ramanathan, and R. Chandra, "BeamSpy: Enabling Robust 60 GHz Links Under Blockage," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016.

[33] "Wilocity 802.11ad Multi-Gigabit Wireless Chipset," http://wilocity.com, 2013.

[34] D. Halperin, W. Hu, A. Sheth, and D. Wetherall, "Tool Release: Gathering 802.11n Traces with Channel State Information," *ACM SIGCOMM CCR*, 2011.

[35] J. Wang, Z. Lan, C.-S. Sum, C.-W. Pyo, J. Gao, T. Baykas, A. Rahman, R. Funada, F. Kojima, I. Lakkis *et al.*, "Beamforming Codebook Design and Performance Evaluation for 60GHz Wideband WPANs," in *IEEE Trans. on Vehicular Technology*, 2009.

[36] T. S. Rappaport, R. W. H. Jr., R. C. Daniels, and J. N. Murdock, *Millimeter Wave Wireless Communications*. Prentice Hall, 2014.

[37] M. H. Hayes, *Statistical Digital Signal Processing and Modeling*. John Wiley & Sons, 2009.

[38] B. D. Van Veen and K. M. Buckley, "Beamforming: A Versatile Approach to Spatial Filtering," *IEEE ASSP Magazine*, 1988.

[39] I. Stevanovic, A. Skrivervik, and J. R. Mosig, "Smart Antenna Systems for Mobile Communications," Tech. Rep., 2003.

[40] K. Tan, J. Fang, Y. Zhang, S. Chen, L. Shi, J. Zhang, and Y. Zhang, "Fine-Grained Channel Access in Wireless LAN," *ACM SIGCOMM*, 2011.

[41] J. P. Ballantine and A. R. Jerbert, "Distance from a Line, or Plane, to a Poin," *Mathematical Association of America*, 1952.
[42] T. Wei and X. Zhang, "mTrack: High-Precision Passive Tracking Using Millimeter Wave Radios," in *Proc. of ACM MobiCom*, 2015.
[43] D. Arthur and S. Vassilvitskii, "K-means++: The Advantages of Careful Seeding," in *ACM-SIAM Symposium on Discrete Algorithms*, 2007.
[44] A. Maltsev, R. Maslennikov, A. Sevastyanov, A. Lomayev, and A. Khoryaev, "Statistical Channel Model for 60 GHz WLAN Systems in Conference Room Environment," in *Antennas and Propagation (EuCAP), 2010 Proceedings of the Fourth European Conference on*. IEEE, 2010.
[45] Degli-Esposti and etc., "Ray-Tracing-Based mm-Wave Beamforming Assessment," *IEEE Access*, 2014.
[46] Y. Ji, S. Biaz, S. Pandey, and P. Agrawal, "ARIADNE: A Dynamic Indoor Signal Map Construction and Localization System," in *Proc. of ACM MobiSys*, 2006.
[47] B. Neekzad, K. Sayrafian-Pour, J. Perez, and J. S. Baras, "Comparison of Ray Tracing Simulations and Millimeter Wave Channel Sounding Measurements," in *Proc. of IEEE PIMRC*, 2007.
[48] A. Khafaji, R. Saadane, J. El Abbadi, and M. Belkasmi, "Ray Tracing Technique based 60 GHz Band Propagation Modeling and Influence of People Shadowing," *IJECSE*, 2008.
[49] E. De Groot, T. Bose, C. Cooper, and M. Kruse, "Remote Transmitter Tracking with Raytraced Fingerprint Database," in *IEEE MILCOM*, 2014.
[50] Rice University, "Wireless Open-Access Research Platform," http://warp.rice.edu/trac/wiki, 2013.
[51] Pasternack Inc., "60 GHz Transmit/Receive (Tx/Rx) Development System," http://www.pasternack.com/60-ghz-test-development-system-pem003-kit-p.aspx, 2015.
[52] J. Zhang, X. Zhang, P. Kulkarni, and P. Ramanathan, "OpenMili: A 60 GHz Software Radio Platform with a Reconfigurable Phased-array Antenna," in *Proc. of ACM MobiCom*, 2016.
[53] MI-WAVE Inc., "Omni-Directional Antenna Series 267," http://www.miwv.com/millimeter-wave-products/antenna-products/, 2014.
[54] "Axis360 Motion Control System," http://cinetics.com/two-axis360/.
[55] F. Adib, S. Kumar, O. Aryan, S. Gollakota, and D. Katabi, "Interference Alignment by Motion," in *ACM MobiCom*, 2013.
[56] S. Kumar, S. Gil, D. Katabi, and D. Rus, "Accurate Indoor Localization with Zero Start-up Cost," in *ACM MobiCom*, 2014.
[57] F. Gulbrandsen, "Design and Analysis of an X-band Phased Array Patch Antenna," Master's thesis, Norwegian University of Science and Technology, 2013.
[58] Pasternack Inc., "WR-15 Waveguide Horn Antenna Operating From 56 GHz to 66 GHz," http://www.pasternack.com/horn-antenna-50-75-ghz-nominal-34-dbi-gain-wr-15-pe9881-34-p.aspx, 2015.
[59] B. Langen, G. Lober, and W. Herzig, "Reflection and Transmission Behaviour of Building Materials at 60 GHz," in *Proc. of IEEE PIMRC*, 1994.
[60] S. Collonge, G. Zaharia, and G. Zein, "Influence of the Human Activity on Wide-Band Characteristics of the 60 GHz Indoor Radio Channel," *IEEE Trans. on Wireless Comm.*, 2004.
[61] S. Alekseev, A. Radzievsky, M. Logani, and M. Ziskin, "Millimeter Wave Dosimetry of Human Skin," in *Bioelectromagnetics*, 2008.
[62] C. Anderson and T. Rappaport, "In-Building Wideband Partition Loss Measurements at 2.5 and 60 GHz," *IEEE Transactions on Wireless Communications*, 2004.
[63] T. Rappaport, F. Gutierrez, E. Ben-Dor, J. Murdock, Y. Qiao, and J. Tamir, "Broadband Millimeter-Wave Propagation Measurements and Models Using Adaptive-Beam Antennas for Outdoor Urban Cellular Communications," *IEEE Transactions on Antennas and Propagation*, 2013.

Skip-Correlation for Multi-Power Wireless Carrier Sensing

Romil Bhardwaj, Krishna Chintalapudi, and Ramachandran Ramjee

Microsoft Research

Abstract

Carrier sensing is a key mechanism that enables decentralized sharing of unlicensed spectrum. However, carrier sensing in its current form is fundamentally unsuitable when devices transmit at different power levels, a scenario increasingly common given the diversity of Wi-Fi APs in the market and the need for Wi-Fi's co-existence with new upcoming standards such as LAA/LWA. The primary contribution of this paper is a novel carrier sensing mechanism – skip correlation – that extends carrier sensing to accommodate multiple transmit power levels. Through an FPGA based implementation on the WARP platform, we demonstrate the effectiveness of our technique in a variety of scenarios including support for backward compatibility.

1 Introduction

The number of public Wi-Fi hotspots worldwide is over 50 Million and growing rapidly [11]. Increasingly, many hot-spots, often deemed "extended range", transmit at up to 10× higher power than most relatively cheap home routers. For example, a Cisco Aironet 1570 Series hotspot [1] transmits at the maximum FCC allowed power of 1W (30dBm) while the Airport Express AP [2] transmits at 100mW (20dBm).

We motivate this paper by asking, "how does a high-power hot-spot affect neighboring low-power APs?" We emulate this scenario using two identical Wi-Fi APs (TP-LINK Archer C7), 1W-AP and 100mW-AP, with their transmit powers set to 1W and 100mW respectively, and connected to identical laptops, L1 and L100. The APs and their respective clients are placed at two different non-line-of-sight locations separated by walls but operate on the same 5 GHz channel 165 with no other APs present to interfere. First, L100 initiates a TCP download through 100mW-AP and 10 seconds later L1 does the same through 1W-AP. As seen in Figure 1, as soon as L1 initiates its download, L100's throughput drops from 87Mbps to a few Mbps *i.e.,* L100 nearly starves. Once L1 finishes, L100 resumes at full throughput.

This starvation occurs when high-power devices don't sense and back-off to transmissions from low-power devices [14]. To avoid collisions, the Wi-Fi standard requires devices to sense the channel and ascertain that it is "free"before transmitting. A channel is defined to be free if the ambient power is less than -82dBm, deemed *carrier sensing threshold* [9] (CST). In our experiment,

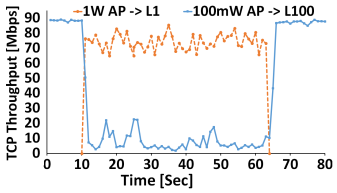


Figure 1: 100mW-AP starvation

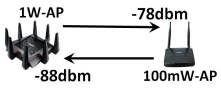


Figure 2: Reason for starvation

100mW-AP received 1W-AP's transmissions at around -78dBm (> CST) (Figure 2). Since 100mW-AP transmitted at 10dB (10×) lower power, 1W-AP received 100mW-AP's transmissions at -88dBm (<CST). Thus, while 100mW-AP sensed 1W-AP's transmissions and backed-off, 1W-AP did not sense and yield to 100mW-AP's transmissions, resulting in near starvation.

In this paper we design a novel carrier sensing mechanism, *skip-correlation*, that addresses the above problem while supporting multiple different power levels. Skip-correlation satisfies the following properties:

- *Sensing symmetry.* A higher power device senses and backs off to all lower (or equal) power devices that sense its transmissions; however it does not sense and back off to transmissions from any other device, as this would result in an unnecessary loss in its throughput.
- *No collateral damage.* Device to device sensing interactions that would not have suffered from sensing asymmetry using existing Wi-Fi standard, remain unaffected.
- *Simple hardware implementation.* Allows easy reuse of existing Wi-Fi carrier sensing digital circuitry.

In Section 2 we show why the first two properties are required in multi-power level scenarios to ensure that while higher power devices share equitably, devices that do no harm are not adversely affected.

Multi-Power Carrier Sensing Threshold Rule. A key observation in our paper is that in order to satisfy the above properties, a sensing device must use a CST value that is *dynamically* determined. Specifically, we show that a sensing device must use a CST value that depends on both, its own transmit power P_{sense}, and that of the transmitting device P_{Xmit}, as captured by the *multi-power CST rule* (Eqns 1a, 1b, Section 2). This dependence of CST on P_{Xmit} poses a key challenge since the sensing device will not know *a priori*, the identity of the transmitter or its power, P_{Xmit}. Further, as per the Wi-Fi standard, carrier sensing must occur within the first 4μs of the packet transmission [19], long before packet

decoding begins. In Section 3, we consider several alternatives that satisfy the Multi-Power CST rule but fall short in terms of their implementation viability.

Skip-Correlation. For carrier sensing, Wi-Fi receivers correlate against a standard preamble that is transmitted at the beginning of each packet. The channel is deemed busy if the correlation value is greater than a threshold. Skip-Correlation relies on the property of correlation based sensing that *doubling the number of samples used in correlation is equivalent to using a CST that is 3dB lower (Section 4)*. In Skip-correlation, while the highest power device correlates against the entire preamble (to allow for lowest CST value), lower power devices skip certain parts of the preamble and correlate against lesser samples. A reverse trend applies to the preambles being transmitted – while the lowest power transmitter transmits the entire preamble (to allow being sensed by the highest power device), higher power transmitters transmit only a part of the preamble. The preamble parts to be transmitted and those to be used in correlation while sensing depend on the transmit power of the device. These parts are carefully chosen to satisfy the Multi-power CST rule, enabling higher power devices to defer to far away lower power devices (avoiding starvation) while simultaneously allowing lower power devices to not defer to other low pow far away devices (avoiding collateral damage).

Hardware Implementation and Backward Compatibility. *Skip-correlation* can be implemented as a minor hardware circuit modification to the popular Schmidl-Cox correlator used in Wi-Fi today and requires only a few additional delay elements and adders. We have extended WARP's existing Wi-Fi implementation to incorporate skip-correlation (Section 6). We demonstrate that *skip correlation implemented on up to 6dB higher power devices can protect unmodified 100mW devices*, thus, supporting backward compatibility. Note that the 6dB limitation only applies when backward compatibility is desired. Skip-correlation technique does not have this limitation when green field preambles are used. We demonstrate the effectiveness of skip-correlation for general multi-power carrier sensing among nodes with 9dB power difference using a green field preamble(Section 7).

Key Difference from Prior Work. Prior related work accepts the lack of carrier sensing symmetry as a given and optimizes the overall network by managing the various nodes' transmit powers, rates and/or CST values (Section 8). However, this won't work in the earlier scenario of an independent hotspot operator impacting the home AP. In this paper, we take a fundamentally different approach and *ensure carrier sensing symmetry in a fully decentralized manner at the physical layer, while allowing nodes to independently choose their transmit powers*.

Non-Wi-Fi scenarios. Skip-Correlation is a general technique and can be used in non-Wi-Fi scenarios.

Sharing with LTE-U, LAA, LWA: New cellular standards such as LTE-U, LAA, and LWA [10] are being designed for unlicensed use alongside Wi-Fi. LAA and LWA use carrier sensing but starvation can occur if a 1W LAA/LWA base-station transmits near a home user's 100mW Wi-Fi AP.

White Spaces: "White spaces" or unused T.V channels are available for unlicensed use in the U.S., U.K., Canada and Singapore [4]. Mobile white space devices, limited to transmit at up to 100mW, may starve near 1W fixed base-stations [25].

Summary of our contributions.

- We propose a new *Multi-Power Carrier Sensing Threshold Rule*, that can eliminate starvation for low power devices while not causing collateral damage.
- We propose a novel sensing technique – *Skip-Correlation* – that implements the above sensing rule and renders itself to a simple implementation.
- We implement skip-correlation on WARP FPGA using a small modification to existing Wi-Fi's carrier sensing circuitry that is backward compatible and demonstrate its efficacy through testbed experiments.

2 Multi-Power Carrier Sensing Threshold Rule

In this section we first derive the multi-power level CST rule from first principles. Then, to provide intuition, we demonstrate how not following the rule can lead to undesirable effects with the help of strawmen alternatives.

Sensing Symmetry. Consider devices AP_H and AP_L with transmit powers P_H and P_L, respectively, where $P_H - P_L = \Delta \geq 0$. The circle in Figure 3 depicts the radius where AP_H's transmissions are received at -82dBm. If AP_L uses a CST of -82dBm as per the Wi-Fi standard, it will sense and back-off from AP_H's transmissions only when AP_L is inside the circle. Thus, for sensing symmetry, AP_H should sense AP_L only when AP_L transmits from within the circle. When AP_L is on the circle, AP_H receives AP_L's transmissions at $-82 - \Delta$. Thus, by *using CST*$=-82-\Delta$*dBm, AP_H ensures that it only senses AP_L when AP_L senses it and no more*.

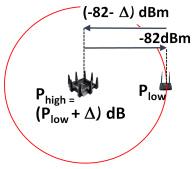


Figure 3: Sensing Symmetry

No Collateral Damage. In the previous discussion we assumed that AP_L used a CST of -82dBm. While there is nothing sacrosanct about -82dBm, it is the value used by the widely adopted Wi-Fi standard. More importantly, we must ensure that the performance of lower power devices that are not causing sensing asymmetry are unaf-

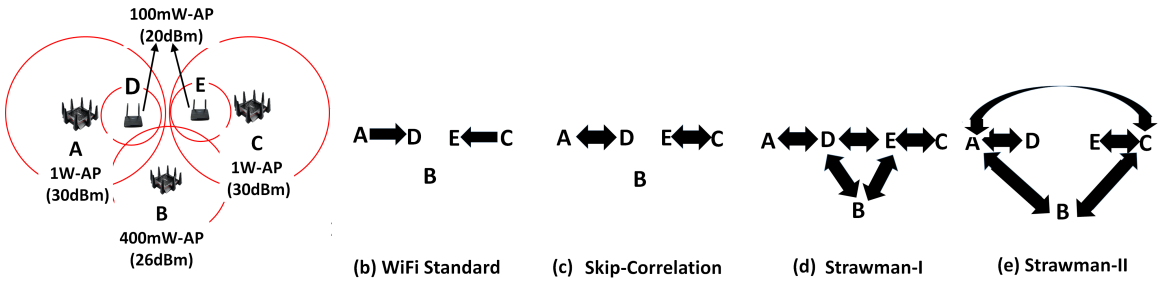


Figure 4: Example showing need for Multi-power CTS Rule

Links	A->D	D<-A	A->B	B->A	D<->E	B->D	D->B	B->E	E->B	E->C	C->E	A<->C	A->E	E->A	C->D	D->C
RSS	-78	-88	-86	-90	-84	-84	-90	-84	-90	-78	-88	-89	-85	-95	-85	-95

Table 1: RSS in dBm for various links in Example in Figure 4

fected. If we were to decrease (or increase) CST for AP_L, it would lead to low power devices unnecessarily backing off (or colliding) with other low power devices, adversely impacting performance. Thus, when the power of a sensing device is less than or equal to that of the transmitting device, we require that the CST value used (e.g, -82dBm) is *unchanged*, thereby ensuring no collateral damage.

Multi-Power Carrier Sensing Threshold Rule. In summary, when device A with power P_A senses a transmission from another device B with power P_B, the CST (in dBm) used by sensing device should be,

$$CST(P_A, P_B) = \begin{cases} -82 & \text{if } P_A \leq P_B, \quad (1a) \\ -82 - (P_A - P_B) & \text{if } P_A > P_B. \quad (1b) \end{cases}$$

As seen from Eqn 1a, 1b, *CST chosen by the sensing device is dynamic and depends both on its own transmit power and that of the transmitting device*. We now show the need for such dependence through an example and contrast it against a few schemes that use fixed CST.

Multi-Power CST Rule vs Fixed Threshold Strawmen. In our example, there are five APs A, B, C, D, E transmitting at 1W, 400mW, 1W, 100mW, and 100mW respectively, as shown in Figure 4(a). Solid circles represent distance at which received power from an AP drop to -82dBm. The received powers for each link is provided in Table 1 for reference. Figures 4(b-e) depict the interference graphs induced due to various carrier sensing schemes used. When the arrow points from X to Y, it indicates that Y senses and backs-off from X.

Wi-Fi Standard. As seen from Figure 4(b), with standard Wi-Fi, there are only two links that can be sensed, *i.e.,* when A transmits to D and when C transmits to E. Consequently, both 100mW devices D and E nearly starve due to 1W devices A and C, respectively.

Skip-Correlation. Skip-correlation uses Eqns 1a,1b to decide the CST for each link. Thus, 1W-AP uses CST -92dBm when a 100mW-AP transmits, -86dBm when 400mW-AP transmits and -82dBm when a 1W-AP transmits. Similarly, 400mW-AP uses CST -82dBm when a 1W-AP or a 400mW-AP transmits and -88dBm when 100mW-AP transmits. As seen from Figure 4(c), *this CST choice based on the transmit powers of both devices* i.e., *transmitting and sensing, ensures that the sensing symmetry is restored between device pairs (A,D) and (E,C) without effecting any other sensing relationship.*

Strawman-I: Preamble Power Equalization. In Strawman-I, all WiFi transmitters somehow ensure that preambles are transmitted at the same power of 1W to make all sensing symmetric.[1] This scheme achieves sensing symmetry since all nodes are equivalent to 1W nodes from a sensing perspective. However, now all the lower power devices B, D and E, with their extended preamble ranges can sense each other (Figure 4(d)). Consequently, they now back-off unnecessarily to each other, resulting in reduced overall throughput.

Strawman-II: Lower Fixed CST for Higher Power Devices. An alternative approach is to make higher power devices more sensitive to lower power receivers. In this scheme, unlike the Multi-Power CST rule, we use a fixed CST value for ease of implementation. The 1W and 400mW APs use Eqn 1b to derive a fixed CST that always protects the lowest power device *i.e.,* 100mW AP. Thus, A and C use a fixed CST of -92dB (-82-10), B uses a fixed CST of -88dBm (-82-6), while D and E use the standard CST of -82dBm. This scheme again ensures sensing symmetry since higher power devices will now always detect all low power devices. However, due to the extended sensing ranges, now the higher power devices A,B, and C sense and back off from each other unnecessarily, resulting in reduced overall capacity.

The above examples demonstrate how using a fixed CST at the receiver or fixed preamble transmit powers at the transmitter causes collateral damage, undermining the throughput of higher or lower power devices depend-

[1] While low power 100mW transmitters are typically not equipped with power amplifiers to transmit at 1W, one could elongate preamble lengths to achieve an equivalent scenario.

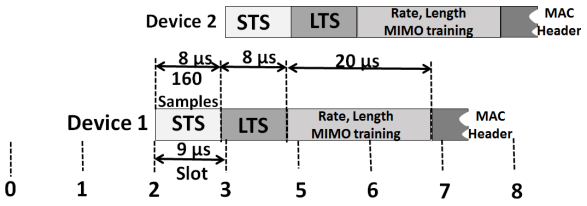


Figure 5: CSMA in WiFi

ing on which class of devices they are chosen to protect. A dynamic CST that depends on the transmit power of both the sensing and the transmitting device is necessary to ensure symmetry without causing collateral damage.

3 Approaches for Implementing Multi-Power CST Rule

The dependence of CST on the transmit power of the transmitting device poses a significant challenge, since identity of the transmitter or its transmit power are unknown *a priori* to the sensing device. To motivate why this is a hard problem that requires a nuanced solution like skip-correlation, we consider two simpler approaches for implementing Multi-Power CST and show why these are not practically viable.

Carrier Sensing: A Primer. Carrier Sensing (CS) in Wi-Fi is achieved by transmitting a special signal, *preamble*, ahead of each packet. During CS, devices continuously try to detect presence of this preamble in the received signal. In Wi-Fi, the preamble used for CS is called the Short Training Sequence (STS) and is 160 samples long (8 μs).

Figure 5 depicts how two devices Device1 and Device2 might share spectrum using CS. First, both devices generate a random counter. In the example in Figure 5, Device1 and Device2 generate 2 and 3, respectively. Every 9μs (Wi-Fi slot-width) the devices decrement their counters. Device1 counts-down to 0 first at the beginning of slot 2 and initiates its transmission. Device2 continues to listen and detects STS from Device1's transmission before the end of Slot 2 and backs-off, avoiding a collision. If Device2 is unable to detect STS from Device1 before the beginning of slot 3, then its counter would have reached 0 and it would transmit its packet causing a packet collision.

3.1 Using Headers

We first consider if one can re-use MAC headers to discover the power of the transmitter. Suppose that every device maintains a table of transmit powers of all its neighbors. The listening device can decode the headers of the received packet to determine the identity of the transmitter and looks up the power of the transmission to make the carrier sensing decision. Alternatively the transmitting device could add an additional field in the headers that encodes transmit power information – this would avoid the need for the lookup. The key drawback of these approaches is that they require decoding of headers before the back-off decision can be taken.

As discussed earlier, the Wi-Fi standard requires back-off decision be taken within 9μs of the packet transmission. The identity of the transmitting device is only known in the MAC header which arrives about 50μs into the packet transmission.

The fastest way for the transmitter to let the receiver know of the transmitter's power level is by adding an additional power level field in the PHY. The earliest this can be done is in the OFDM symbol immediately after the LTS (it cannot be added before LTS as LTS is used for frequency offset estimation – a necessary step before decoding). Thus, the slot-width would have to be at least 20μs (LTS + STS + 1 OFDM Symbol).

Why can't we simply increase the Wi-Fi slot width? Increasing the Wi-Fi slot width to 20μs would result in more than doubling of the average backoff interval, severely reducing MAC efficiency and overall throughput [19].

3.2 Using Orthogonal Preambles

In this approach, a unique preamble is assigned to each power level. Each of these preambles are orthogonal to the others and have low cross-correlation properties (like CDMA codes). Thus, each preamble uniquely encodes the transmit power level information within itself. A device transmits the preamble corresponding to its transmit power level. Listening devices now correlate against all possible preambles simultaneously using a matched-filter (Appendix B) bank. The bank uses a different CST for each correlator given by Eqn 1a, 1b for sensing.

There are two key drawbacks to this approach – high complexity of implementation and need for very long preambles for correctly inferring the power level. Since Schimdl-Cox is oblivious to the exact preamble transmitted, it cannot be used in this scheme as it will be unable to distinguish between the different preambles. As discussed in Section 6.2, for 4 power levels a matched-filter approach requires over 1500 multipliers while Schimdl-Cox and our proposed scheme Skip-Correlation requires only 5. The other drawback arises from the ability to distinguish between different preambles and inferring the correct preamble through cross-correlation. As we demonstrate in Appendix C, for 4 power levels, even at 128 samples long, the incorrect preamble detections are as high as 14%. Using preambles longer than 128 samples will increase Wi-Fi slot width, resulting in low MAC efficiency. Finally, note that these drawbacks are fundamental to this approach and also apply to other implementations such as Tufvesson [26].

4 Adapting CST

Most Wi-Fi receivers perform carrier sensing by correlating against the Wi-Fi preamble. Skip-correlation relies

on a basic property of correlation-based sensing – *doubling the number of samples used in correlation is equivalent to using a CST that is 3dB lower.* In this section we explain this property and provide a proof sketch (detailed proof in Appendix B) for one specific correlation-based scheme – Schmidl-Cox detection – commonly used in Wi-Fi devices today and that we modify in our implementation[2].

Schmidl-Cox Preamble Detection Let $\mathbf{S}(n)$ be the preamble of length L samples that is transmitted. The received signal $\mathbf{S}_{recv}(n)$ received by the sensing device for the cases when no preamble was transmitted and when a preamble was transmitted is given by Eqns 2a, 2b respectively as,

$$\mathbf{S}_{recv}(n) = \begin{cases} \mathbf{N}(n) & \text{no preamble Xmit,} \quad (2a) \\ \mathbf{S}(n) * \mathbf{H} + \mathbf{N}(n) & \text{preamble Xmit.} \quad (2b) \end{cases}$$

In Eqns 2a,2b, $\mathbf{N}(n)$ is the receiver noise, $\mathbf{H}$ is the channel response that captures how the channel transforms the transmitted signal due to effects such as multi-path fading and, $*$ is the convolution operation.

In Schmidl-Cox detection, $\mathbf{S}(n)$ comprises two identical halves *i.e.,* $\mathbf{S}(i) = \mathbf{S}(i + \frac{L}{2})$. As the transmission passes through the channel, each of its halves are effected by the channel in exactly the same way and are expected to be identical. Consequently, the receiver correlates two consecutive $\frac{L}{2}$ length windows of the received signal as,

$$C = \left| \sum_{i=1}^{i=\frac{L}{2}} \mathbf{S}_{recv}(i) \mathbf{S}_{recv}(i + \frac{L}{2}) \right|^2 \quad (3)$$

A high correlation value ($C > C_{th}$) indicates that a preamble was transmitted and the channel is deemed busy. The advantage of using Schmidl-Cox is that it is robust to multi-path effects in the channel.

Detection Probability depends on Product of Received Signal Strength and Correlation Length. The Wi-Fi standard specifies that a signal at -82dBm must be detected with a probability 90% or higher [9]. Receiver noise induces variability in the correlation value C. Choosing a low C_{th} will result in a high detection probability, but also increased false detection rate *i.e.,* concluding that there is a preamble when there is in fact only noise. C_{th} is usually calibrated such that the detection probability is 90% (or some higher value) when the received signal strength is at -82dBm. Appendix B describes in detail, how to choose C_{th}. As shown in Appendix B, for the operating regime of typical Wi-Fi receivers, the preamble detection probability is governed by a χ^2_1 distribution with mean,

$$\mu(C) \approx (LE_S)^2/4 \quad (4)$$

where E_S is received signal strength of the preamble. Eqn 4 implies that the probability distribution and hence the detection probability remain approximately unaltered if the product LE_S is kept the same.

[2]The property holds for other correlation based schemes as well.

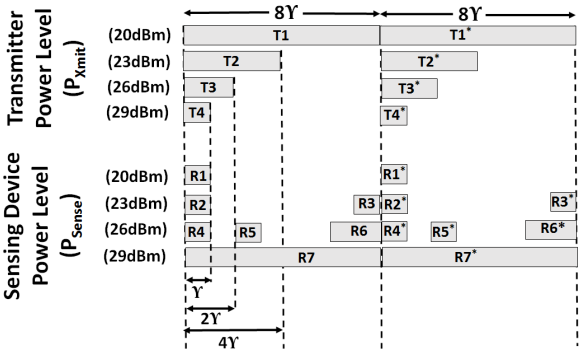


Figure 6: A Skip Correlation Example

Adapting CST by changing L. Suppose that when the received preamble's signal strength is E_S = -82dBm and the number of preamble samples used for correlation is L, then preambles are detected with 90% probability. By using $2L$ samples for correlation instead of L, the same detection probability would be achieved when E_S is at -85dBm (half of -82dBm). Thus, Eqn 4 can be interpreted as doubling the number of samples used in the correlation will allow detection at a received signal strength 3dB lower with the same detection probability *i.e.,* lowering the CST by 3dB.

5 Skip-Correlation

In this section we describe our proposed scheme – *Skip-Correlation* – that implements carrier sensing according to the Multi-Power CST rule. Our scheme reuses Wi-Fi's preamble (STS) and most of Wi-Fi's carrier sensing circuits while adding very little extra circuitry.

We first provide the reader intuition for skip-correlation using an example. Then we formally describe skip-correlation for N arbitrary power levels, followed by a proof of how it satisfies the Multi-Power CST rule.

P_{Sense} \ P_{Xmit}	20dBm	23dBm	26dBm	29dBm
	L (CST)	L (CST)	L (CST)	L (CST)
20dBm	2γ (-82)	2γ (-82)	2γ (-82)	2γ (-82)
23dBm	**4γ (-85)**	2γ (-82)	2γ (-82)	2γ (-82)
26dBm	**8γ (-88)**	**4γ (-85)**	2γ (-82)	2γ (-82)
29dBm	**16γ (-91)**	**8γ (-88)**	**4γ (-85)**	2γ (-82)

Table 2: Number of samples used in correlation (L) and CST (in dBm) for all combinations of power levels

5.1 A Skip-Correlation Example

Figure 6 depicts an example with four different power levels 20, 23, 26, 29dBm with corresponding parts of the preamble transmitted and correlated against. The entire preamble is 16γ samples long with two identical halves of 8γ samples to allow Schmidl-Cox detection. We assume that γ is chosen such that the correlation length of 2γ samples corresponds to a CST of -82dBm, *i.e.,* it

provides a 90% detection probability when the received signal strength is -82dBm.

For each of the 16 transmitter power level (P_{Xmit}) and sensing device power level (P_{Sense}) combinations, Table 2 lists the number of preamble samples used for correlation (L) and the corresponding CST (in dBm) used by the sensing device. One can verify that all 16 combinations satisfy the Multi-power CST rule. As seen from Table 2, CST is reduced only for 6/16 scenarios (marked as bold) by the necessary amount dictated by the Multi-Power CST rule to eliminate starvation but remains at -82dBm for the rest 10/16 scenarios to avoid collateral damage. We now describe three combinations (gray cells in Table 2) to provide intuition.

20dBm transmitting, 20dBm sensing. While the 20dBm transmitter transmits all 16γ samples (T1 and T1'), the 20dBm sensing node only correlates 2γ samples (R1 and R1'). Thus, the 20dBm sensing device uses a CST of -82dBm as desired.

20dBm transmitting, 29dBm sensing. The 29dBm receiver correlates against all 16γ samples (R7 and R7') *i.e.,* 8 times more samples than the 20dBm receiver. As discussed in Section 4, this means that the CST for this pair is 8 times (9dB) lower than -82dBm or -91dBm.

23dBm transmitting, 26dBm sensing. The 23dBm transmitter transmits preamble parts T2 and T2'. The 26dBm receiver correlates against pieces R4-R6 and R4'-R6'. T2 and T2', however, do not overlap with R6 and R6'. Thus no preamble samples are transmitted for R6 and R6' to correlate upon. Consequently, correlation occurs only over the 4γ samples R4, R5, R4' and R5'. Thus, the CST for this pair is 2 times (3dB) lower than -82dBm or -85dBm.

5.2 Notation Used

We now outline the notation used for describing the parts of preambles to be transmitted and correlated against for a general scenario with N power levels.

- The Preamble : The entire preamble has L samples and is denoted by $\mathbf{S} = [\mathbf{S}_1, \mathbf{S}_2, \cdots, \mathbf{S}_L]$ with its left half $\mathbf{S}^{LH}$ identical to its right half $\mathbf{S}^{RH}$.
- Transmitted Preamble: The preamble transmitted by a device with power level P comprising multiple pieces is denoted by $\mathbf{S}_P^{Xmit}$ with its right and left halves as $\mathbf{S}_P^{Xmit,LH}$ and $\mathbf{S}_P^{Xmit,RH}$.
- Correlated Preamble: The preamble correlated against is denoted by $\mathbf{S}_P^{Recv}$ comprising $\mathbf{S}_P^{Recv,LH}$ and $\mathbf{S}_P^{Recv,RH}$ as its identical left and right halves.
- Preamble Part: A part of the preamble from sample index i to j is denoted as $\mathbf{S}[i,k] = [\mathbf{S}_i, \mathbf{S}_j, \cdots, \mathbf{S}_k]$.
- Union : $\mathbf{C} = \mathbf{A} \cup \mathbf{B}$, indicates that a preamble $\mathbf{C}$ contains pieces from both preambles $\mathbf{A}$ and $\mathbf{B}$.

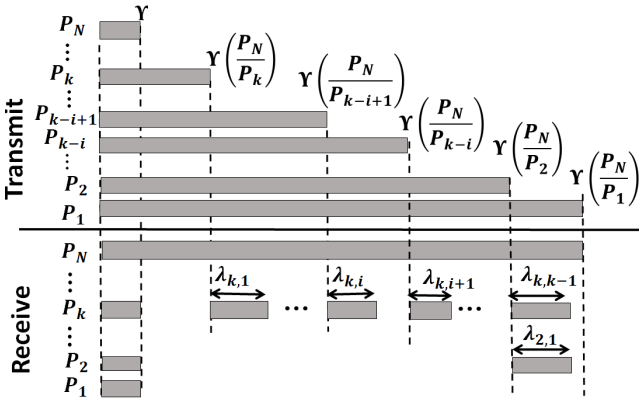


Figure 7: Skip Correlation for $P_1 < P_2 < \cdots < P_N$

- Intersection: $\mathbf{A} \cap \mathbf{B}$ denotes pieces that are common to both $\mathbf{A}$ and $\mathbf{B}$.

5.3 Arbitrary Transmit Powers

In this section we describe Skip-Correlation for general scenarios where devices may choose from N distinct power levels $P_1 < P_2 < \cdots < P_N$.

For the following discussion, we assume that any subpart of the preamble with γ samples along with its identical copy *e.g.,*$\mathbf{S}[1,\gamma]$ and $\mathbf{S}[\frac{L}{2}+1, \frac{L}{2}+\gamma]$, ensures detection at -82 dBm (or some other standard specified value). The entire preamble $\mathbf{S}$ has $L = 2\gamma\frac{P_N}{P_1}$ samples with its first half identical to its second half. Figure 7 depicts the first half of the preambles transmitted and correlated by various devices (the second half is identical).

Preamble Transmission. A device with transmit power P_k transmits $\mathbf{S}_{P_k}^{Xmit,LH}$ comprising the first $\gamma\frac{P_N}{P_k}$ of the preamble and its copy in the second half *i.e.,*

$$\mathbf{S}_{P_k}^{Xmit,LH} = \mathbf{S}\left[1, \gamma\frac{P_N}{P_k}\right] \tag{5}$$

This ensures that devices of all power levels transmit the same amount of total preamble energy of $2\gamma P_N$.

Preamble Reception. A device with transmit power level P_k correlates against k different pieces of the preamble in the left half captured in $\mathbf{S}_{P_k}^{Recv,LH}$ and its shifted copy in the second half, *i.e.,*

$$\mathbf{S}_{P_k}^{Recv,LH} = \mathbf{S}[1,\gamma] \cup \bigcup_{i=1}^{i=k-1} \mathbf{S}\left[\gamma\frac{P_N}{P_{k-i+1}}, \gamma\frac{P_N}{P_{k-i+1}} + \lambda_{k,i}\right] \tag{6a}$$

$$\lambda_{k,i} = P_k\gamma\left(\frac{1}{P_{k-i}} - \frac{1}{P_{k-i+1}}\right) \tag{6b}$$

The first piece is $\mathbf{S}[1,\gamma]$. The rest $k-1$ pieces, have lengths $\lambda_{k,i}, i = 1, \cdots, k-1$ starting at sample number $\gamma\frac{P_N}{P_{k-i+1}}$.

Proof of Correctness. When a device transmits at a power level P_i to a device with transmit power level P_k, the following properties P1 and P2 always hold,

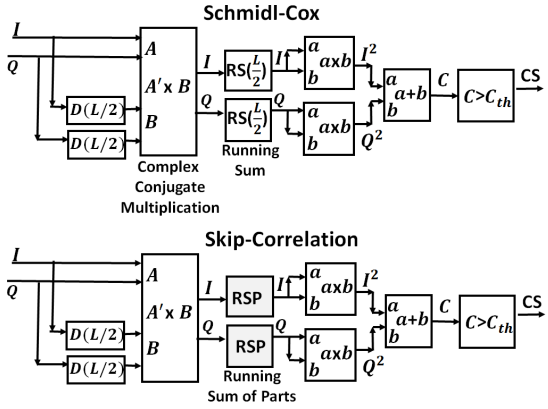


Figure 8: High Level View of Changes between Standard Schmidl-Cox and Skip-Correlation

- P1 : if $P_i \geq P_k$ the correlation is only against $\mathbf{S}[1, \gamma]$, which is 2γ samples corresponding to a CST of -82dBm.
- P2 : if $P_i < P_k$ the number of preamble samples correlated at the receiving device (in $\mathbf{S}_{P_i}^{Xmit} \cap \mathbf{S}_{P_k}^{Recv}$) is $2\frac{P_k}{P_i}\gamma$ – corresponding to a CST that is $\frac{P_k}{P_i}$ lower than -82dBm *i.e.,* $-82-(P_k dB - P_i dB)$ in log-scale.

Thus, CST used in skip-correlation matches the Multi-Power CST rule. The proof of these properties is provided in Appendix A.

6 Design and Implementation

Schmidl-Cox detection is typically implemented on an ASIC in the Wi-Fi card as a digital circuit. Skip-Correlation can be realized through a minor modification of Schmidl-Cox. Hence, we start by describing how Schimdl-Cox correlation is implemented and then explain how to modify it to implement Skip-Correlation.

6.1 Skip-Correlation Circuit Design

The block-diagram of Schmidl-Cox detector is provided in Figure 8. The received signal is a sequence of complex samples with I (real) and Q (imaginary) parts. $D(x)$ blocks are delay elements that introduce a delay of x samples and usually implemented using FIFO of depth x elements. The complex conjugate multiply block takes two complex samples $s(i)$ and $s(i + \frac{L}{2})$ to perform the operation $s(i)'s(i + \frac{L}{2})$, where $'$ denotes the complex conjugate operation. The real and imaginary parts from the complex conjugate multiplier are each fed into an running sum block (described later in this section) that maintains the sum of the last $\frac{L}{2}$ samples. Finally, the real and imaginary parts of the running sum are squared and added to be compared against the threshold C_{th}.

The only change required to implement Skip-Correlation is to replace the running-sum (RS) block of Schmidl-Cox by a running sum of parts (RSP in Figure 8).

Circuit for Running Sum used in Schmidl-Cox The

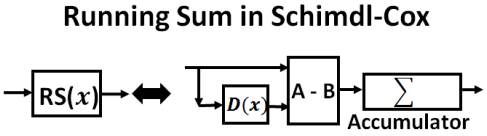


Figure 9: Running sum block in Schmidl-Cox with History Length x.

running sum over the past x samples in Schmidl-Cox is implemented by the following equation,

$$sum(i+1) = [s(i+1) - s(i-x)] + sum(i) \quad (7)$$

The circuit implementation for a running sum $RS(x)$ is provided in Figure 9. A-B block subtracts B from A, and this case produces $s(i+1) - s(i-x)$ and $\sum$ is an accumulator that continuously adds its inputs.

Circuit for Running Sum of Parts in Skip-Correlator. The circuit for running sum of preamble parts is fundamentally the sum of several running sums, one for each piece of the preamble and is given in Figure 10. Using the same notation used in Section 5.3, for a device with power level P_k the Skip-Correlator is a cascade of k running sum blocks and delay elements. The only exception is P_N which correlates the entire preamble and is simply $RS(\frac{L}{2})$.

6.2 Circuit Complexity

In this section we analyze the circuit complexity of implementing the Skip-Correlator and the orthogonal preambles proposal (Section 3.2) and compare them against Schmidl-Cox. Since multipliers and adders constitute the most expensive elements in these circuits we measure complexity by the number of multipliers and adders required.

Schmidl-Cox. In the overall Schmidl-Cox circuit there is one complex conjugate multiplication (3 multipliers and 4 adders [17]), two RS blocks (2 adders in each), two multipliers for squaring real and imaginary parts and finally another adder – a total of *5 multipliers and 9 adders*.

Skip-Correlation With N power levels. The only change in implementing the Skip-Correlator is the RSP block. A receiver with power level P_k has $2k$ RS blocks (each with 2 adders). Thus, the complexity is given by 5 multipliers and 4k+5 adders. The device at P_{N-1} will have the greatest complexity at 5 multipliers and $4N+1$ adders. For 4 power levels this corresponds to 5 multipliers and 17 adders.

Orthogonal Preambles - Multiple Matched Filters. Unlike Schmidl-Cox, correlation against a matched filter (Appendix B) cannot be implemented using a running sum. Thus, it requires L complex multipliers for computing each of L products and finally $\frac{L}{2}$ adders for computing the sum of the products using a binary tree. Two

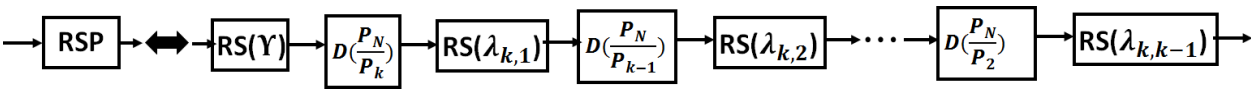


Figure 10: Circuit for Running Sum of Parts in Skip-Correlator.

multipliers are needed for squaring the real and imaginary parts and finally an adder to compute C. This adds up to $4.5L + 1$ adders and $3L + 2$ multipliers. If there are N power levels, there will be N matched filters in the correlation bank leading to $3NL + 2$ multipliers and $4.5NL + 1$ adders. As shown in Section B, for $N = 4$, we will need $L = 128$ in order to achieve a false positive rate $< 10^{-3}$. Thus, the circuit for orthogonal preambles will need over 1500 multipliers and 2300 adders. This demonstrates that implementing the orthogonal preamble approach is not practically viable and has at least two orders of magnitude higher complexity.

6.3 Implementation on WARP

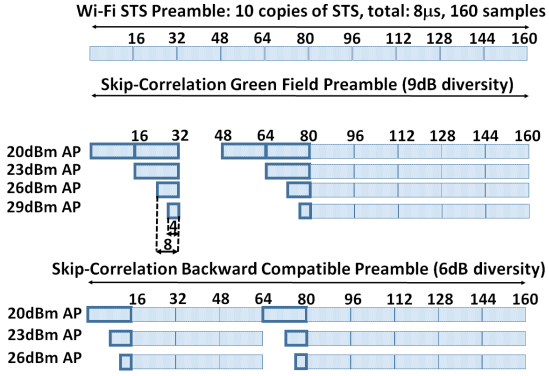


Figure 11: Green Field and Backward Compatible Skip-Correlation Preambles Transmitted by Various Nodes

The WARP software defined radio platform provides an implementation of Wi-Fi with Schmidl-Cox correlation for CS. We modified this Schimdl-Cox as shown in Figure 8 to implement a four power level skip-correlation.

Correlation Lengths Used. Wi-Fi standard requires that carrier sensing be performed within the first 4μs [19] of the 9μs slot. Since most modern day commercial Wi-Fi radios have a noise floor of -91dBm or less [3, 7, 8], we consider 0dB SNR as corresponding to -91dBm. Thus, 9dB SNR corresponds to -82dBm. As discussed on Section B, at 9dB SNR, $L = 8$ samples suffices for Schmidl-Cox to give a false positive rate below 10^{-6}; thus for our Skip-Correlation we choose $\gamma = 4$. The full preamble length for supporting 9dB power difference is $L = 8 * 2 * \gamma = 64$. Thus, we use only 64 of the 160 samples of the WiFi STS for skip-correlation sensing.

Skip-Correlation Green Field Preambles. Figure 11 depicts the Green Field preambles used for transmitting for each of the four different power levels. Note that Wi-Fi has other preambles (e.g., LTS, MIMO preambles etc.) that come after the STS – these are unaffected and hence not shown. In our design we reuse the WiFi-STS as much as possible. Only the highlighted parts of the STS preamble are used for carrier sensing in skip-correlation. The rest of the preamble is used for other functions such as AGC gain estimation and DC-offset correction. In our trials we found that it was necessary for the even the 100mW device required a gap of 16 samples between its two halves. This is because, in absence of the gap, multipath from previous samples distorts the second half. This addition of zeros requires changing the delay element $D(\frac{L}{2})$ in Figure 8 to $D\big(\frac{L}{2} + 16\big)$. However, even with zeros, skip-correlation carrier sensing occurs within first 80 samples (4μs).

Skip-Correlation Backward Compatible Preambles. Given the large number of 100mW legacy devices, we want Skip-correlation to work in a backward compatible mode in the presence of legacy 100mW devices. Through experimental trials we found that several legacy phones and low power routers did not work with the Green Field Preamble shown in Figure 11. Our trials indicated that many legacy devices require additional 2-3 contiguous STS sequences within the first 4μs to function correctly. We guess that, these devices take longer time for AGC stabilization and DC-offset correction. Thus, our Backward Compatible WiFi STS preamble (Figure 11), introduces 3 additional STS sequences within the first 4μs. However, with this change we can only support up to 6dB power diversity. Thus, in our this mode we choose only three power levels 20dBm, 23dBm and 26dBm.

7 Testbed Results

We first present experimental results demonstrating the effectiveness of skip-correlation in a backward compatibility setting with legacy low power Wi-Fi routers and smartphones, for power differences of up to 6dB. We then present a detailed analysis of how skip-correlation ensures symmetry when four power levels spanning 9dB in power difference co-exist. All experiments were conducted on an unused 20 MHz Wi-Fi channel in the 5 GHz band.

7.1 Backward Compatibility Experiments

In this section, we use the three settings shown in Figure 12 to answer the question: *how well does skip-correlation work in the presence of legacy Wi-Fi APs and phones?*

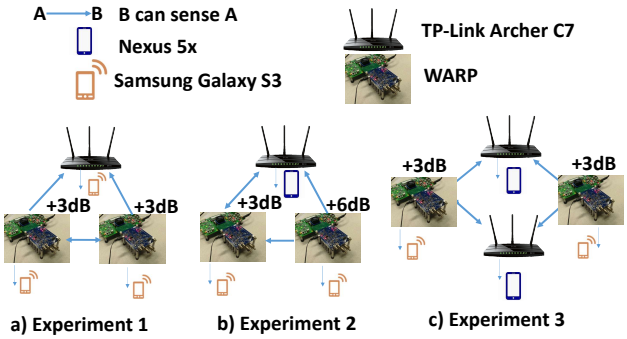


Figure 12: Experimental setup with legacy Wi-Fi devices

In all the experiments, we use a mix of off-the-shelf TP-Link Archer C7 APs and WARP APs, each associated with one smartphone, and performing an iperf udp download to the smartphone (tcp results are qualitatively similar). The Archer C7 APs use auto-rate and MIMO (2X2) while the WARP APs are set to operate at a fixed PHY rate of 6 Mbps. Each node is only aware of its own configured transmit power and has no explicit information about the transmit power or other information of any other nodes. In isolation, the TP-Link client gets an average UDP throughput of 86 Mbps while the WARP clients get an average UDP throughput of 4.4 Mbps.

In the first setting (Figure 12(a)), there are 3 APs and 3 smartphones, where one of the APs is TP-Link Archer C7 router and the other two APs are WARP devices. The two WARP APs act as high power APs with 3dB higher power than the TP-Link AP. The nodes are placed such that, when all APs run the standard Wi-Fi, the WARP nodes are unable to sense the TP-Link AP while the TP-Link AP can sense the WARP nodes. Further, the WARP nodes are able to sense each other. At time $t = 0$s, the TP-Link AP starts the download to its client and at time $t = 20$s, the two high-power WARP APs start their respective downloads. Finally, at $t = 180$s, the high-power APs stop their download.

The results without and with skip-correlation are shown in Figures 13 and 14, respectively. The y-axis on the left shows the throughput for clients connected to the fixed rate WARP AP while the y-axis on the right shows the throughput for client(s) connected to TP-Link AP. With standard Wi-Fi, one can clearly see in Figure 13 that the client connected to TP-Link AP nearly starves – 0.94 Mbps average throughput whenever the high power WARP APs are in operation while the WARP clients each get about 2.15Mbps (or about half the time-share). With Skip-correlation, the starvation is eliminated in Figure 14 and the three nodes share approximately equally in time – the low power client gets an average throughput of 24 Mbps (28% of 86 Mbps, about one-third) while the clients connected to WARP get 1.4 Mbps each on average (32% of 4.4 Mbps, about one-third).

The second experiment (Figure 12(b)) also has three APs but with three different power levels: the TP-Link AP is the low power AP and the two WARP APs are at 3dB and 6dB higher power, respectively. The nodes are placed such that, when all APs run the standard Wi-Fi, the lower power APs are able to sense the higher power APs, but not vice-versa. At time $t = 0$s, the TP-Link AP starts the download to its client and at time $t = 20$s, the two higher power WARP APs start their respective downloads. Finally, at $t = 180$s, the higher power APs stop their download.

The results without and with skip-correlation are shown in Figures 15 and 16, respectively. With standard Wi-Fi, one can clearly see in Figure 15 that the two clients connected to the TP-Link AP and +3dB WARP APs mostly starve (average throughputs of 1.3Mbps and 0.24Mbps, respectively) whenever the +6dB WARP AP is in operation (whose client gets average throughput of 4.3 Mbps). With Skip-correlation, the starvation of both the TP-Link AP and +3dB WARP APs are eliminated in Figure 16, and all three clients again get about one-third the time-share (average throughputs of 24.2 Mbps, 1.36 Mbps and 1.44 Mbps).

In the third experiment (Figure 12(c)), there are 4 APs and 4 smartphones, where two of the APs are TP-Link Archer C7 and the other two APs are WARP devices. The two WARP APs act as high power APs with 3dB higher power than the TP-Link APs. The nodes are placed such that, when all APs run standard Wi-Fi, the WARP APs are unable to sense the TP-Link APs while the TP-Link APs can sense the WARP APs. Further, the two WARP APs cannot hear each other. At time $t = 0$s and $t = 15$s, the two TP-Link APs start their download to their respective clients and at time $t = 30$s and $t = 45$s, the two high-power WARP APs start their respective downloads. Finally, at $t = 180$s, the high-power APs stop their download.

The results without and with skip-correlation are shown in Figures 17 and 18, respectively. With standard Wi-Fi, as seen in Figure 17, the two clients connected to the low-power TP-Link APs starve (average throughputs of 1 Mbps and 1.8 Mbps) whenever the WARP APs with 3dB higher power are in operation (WARP clients get average throughputs of 4.3 Mbps each). With skip-correlation, the starvation is eliminated in Figure 18, with all four clients getting one-half time-share each (average throughputs of 39 Mbps, 43 Mbps, 2.1 Mbps and 2.2 Mbps).

7.2 Green Field: Micro-benchmarks

We now look at what happens with skip-correlation under-the-hood. For these experiments we support four power levels 20dBm, 23dBm, 26dBm and 29dBm and consequently use the green field preamble patterns

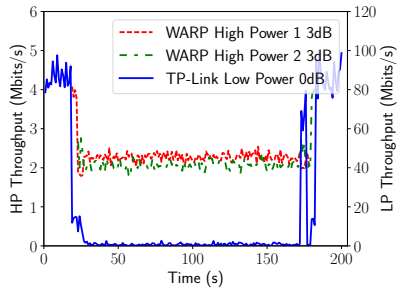


Figure 13: Standard Wi-Fi: Expt. 1

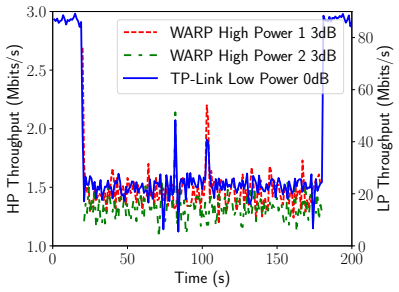


Figure 14: Skip-Correlation: Expt. 1

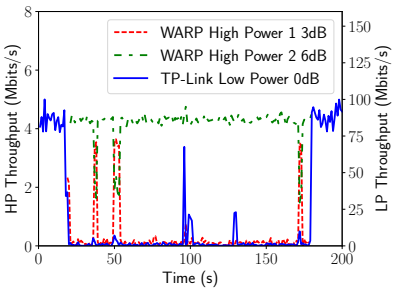


Figure 15: Standard Wi-Fi: Expt. 2

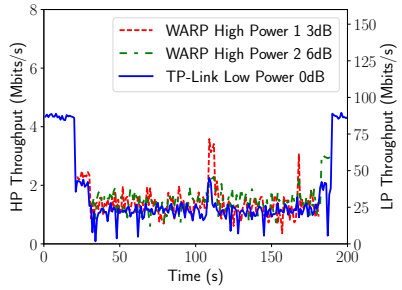


Figure 16: Skip-Correlation: Expt. 2

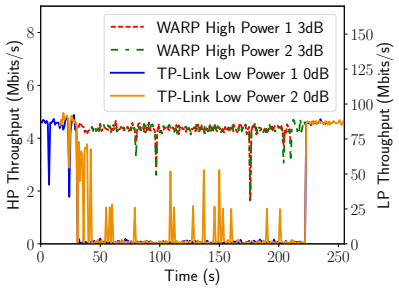


Figure 17: Standard Wi-Fi: Expt. 3

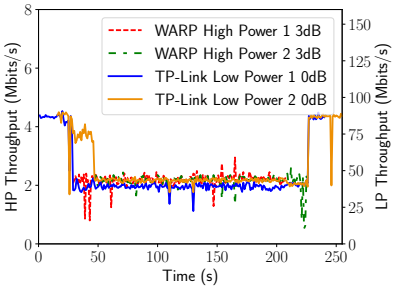


Figure 18: Skip-Correlation: Expt. 3

shown in Figure 6 implemented on WARP nodes. We use received SNRs to calibrate our experiments. Our goal is to demonstrate that skip-correlation ensures carrier sensing symmetry for all pairs of devices.

7.2.1 Establishing C_{th}

C_{th} is determined from the false negative detection curves when the received power is -82dBm (Appendix B). Thus, we placed two WARP nodes in line-of-sight to minimize signal variation and adjusted the transmit power of one such that the other received at 9dB SNR (-82dBm). We then collected samples from 30,000 transmissions each separated by 1 second (almost over 9 hours). There was about 5dB variation in received power despite line-of-sight since there were people moving during the course of the experiment. To ensure that the received powers were as close to -82dBm as possible we considered only those transmission that had received SNRs in the range 9 ± 1 dB, which were little over 10000 in number. We then computed $P(C|$ Xmit, $-91dBm, -82dBm)$ for the Skip-Correlator using a histogram with 100 bins over the collected data. We also computed $P(C|$ No Xmit, $-91dBm)$ by correlating against the noise samples. Using these distributions we then computed the false negative curves $\eta_-(C|-91dBm, -82dBm)$ and the false positive curve $\eta_+(C|-91dBm)$ using Eqn 15 and Eqn 16, depicted in Figure 19. The curves are "close" to the theoretically predicted curves in Appendix B, Figure 22. Using the point at $\eta_- = 0.1$ in the curve, we established C_{th} as shown in Figure 22; it was found to be about -15.7, very close to the theoretically predicted value of -15.5 in Figure 22. The false positives corresponding to this threshold are very small. We use this value of C_{th} in all the following experiments.

7.2.2 CS Symmetry

In this experiment, we use two WARP devices A and B to test carrier sensing symmetry. Due to space considerations, three node experimental results are in Appendix D.

We conducted carrier symmetry experiments at four different non-line-of-sight location pairs L-91, L-88,L-85 and L-82. For each location pair, locations of both A and B were changed in a way that for the location pair L-X, the devices were so placed that when one device transmitted at 20dBm transmit power, the other would receive at X dBm. For example, for the location pair L-85, when one device transmitted at 20dBm the other received at -85dBm (6dB SNR). Note that since the experiments lasted a few hours, with people moving about, in reality there was significant variation of the received signal strength at each location. Figure 20 shows the probability distribution of received powers at B when A transmitted at 20dBm power. Thus, there was 5dB-12dB variation in received power across the various locations.

At each location we experimented with 4 different power levels 20dB, 23dBm, 26dBm and 29dBm and obtained data for all the possible 10 unique combinations of transmitter and receiver powers. The experiment at

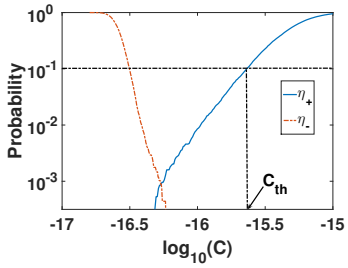


Figure 19: Threshold selection for skip-correlation

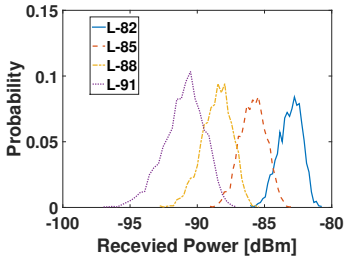


Figure 20: Variation of received signal strength for each of the locations

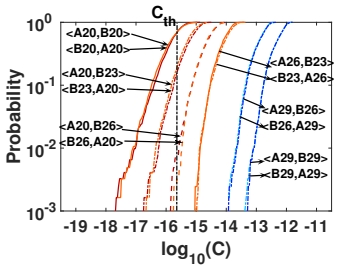


Figure 21: Carrier Sensing Symmetry for Skip-Correlation at location L-85.

Skip-Correlation Detection Probabilities

	A20-B20		A23-B20		A26-B20		A29-B20		A23-B23		A26-B23		A29-B23		A26-B26		A29-B26		A29-B29	
	F	R	F	R	F	R	F	R	F	R	F	R	F	R	F	R	F	R	F	R
L-91	1	0	6	10	45	79	89	97	7	5	46	75	89	98	48	44	89	96	88	88
L-88	2	1	22	25	81	96	98	100	23	19	81	96	98	100	80	78	98	100	98	98
L-85	48	45	92	90	100	100	100	100	92	85	100	100	100	100	100	100	100	100	100	100
L-82	92	91	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100

Schmidl-Cox Detection Probabilities

	A20-B20		A23-B20		A26-B20		A29-B20		A23-B23		A26-B23		A29-B23		A26-B26		A29-B26		A29-B29	
	F	R	F	R	F	R	F	R	F	R	F	R	F	R	F	R	F	R	F	R
L-91	0	0	0	0	3.3	0	**90.5**	**0**	0	0	**90.2**	**0**	**91.2**	**0**	5	3.3	**90.5**	**3.3**	90.5	91.2
L-88	0	0	2.4	0	**90.2**	**0**	**99.8**	**0**	4.5	2.4	**90.3**	**2.4**	**99.9**	**2.4**	90.3	90.2	99.8	90.2	99.8	99.9
L-85	31.9	35.7	**90**	**35.7**	**100**	**31.9**	**100**	**35.6**	90	90.4	99.9	89.9	100	90.4	100	99.9	100	100	100	100
L-82	91.8	89.6	95	91.9	100	91.8	100	95	98	95	100	98	100	95	100	100	100	100	100	100

Table 3: Detection Probability Symmetry for Skip-Correlation

each location consisted of 3000 rounds. In each round, first A transmitted 4 successive transmissions at each of the four power levels. This was immediately followed by B transmitting 4 successive transmissions at all the four power levels. All four possible skip-correlation patterns (Figure 6) for the receiver were available at each device, so that we could obtain all four possible correlation values. We waited 1 second between each round. The hope here was that within each round the channel would not change significantly, whereas between two consecutive rounds the channel would be different. This strategy made sure that we tried all possible power level combinations in both directions over approximately the same channel, so that 3dB increase in transmitted power would mean 3dB increase in the received power as well within a single round. At the same time, across different rounds we could experiment across changing channel conditions, since each experiment lasted about an hour with people moving about on the floor.

Carrier Sensing Symmetry Observations. We used the correlation values from the above experiments and computed $\eta_-(\ C|\sigma_S^2, \sigma_N^2)$ using Eqn 15 for all 16 power level combinations and at all four locations (64 curves in total). Note however, that since the received signal strength was varying in this experiment from round to round, the distribution is computed over the entire range of variation of signal strength and not at any single received power. Figure 21 depicts a few of these curves from the L-85 location. In Figure 21 the notation $< TY, RZ >$ denotes that that node T transmitted at power Y dBm to node R whose transmit power was set to ZdBm. Thus, $< A29, B20 >$ means that A transmitted at 29dBm to B which was using a skip-correlation pattern corresponding to 20dBm transmit power. Similarly, $< B20, A29 >$ represents the same link but when B transmits at 20dBm to A that uses the skip-correlation pattern corresponding to 29dBm. As seen from Figure 21, the η_- curves $< A29, B20 > < B20, A29 >$ almost overlap; this despite the variation in received signal strengths and changing channel conditions at both the devices - thus indicating a strict adherence to carrier sensing symmetry. The same is true for other power level combinations as well and demonstrates how Skip-Correlation ensure that the detection probabilities at either ends at almost identical despite channel changes and signal variations.

Symmetric Detection Probabilities. Using the threshold C_{th}, for each transmission we made a detection decision and computed the detection percentages in both directions for each link at all power level combinations across all the locations. In order to compare with Schmidl-Cox, as the WARP implementation uses $L = 32$, for the comparison to hold meaningful we chose a threshold such that when one device transmits to another at a received power of -82dBm (9dB SNR), the false negatives are at 10%. Table 3 shows all the detection probabilities obtained across all combinations. In Table 3, the notation AX-BY means node A transmitting at power X dBm and node B transmitting at Y dBm. F and R indicate forward and reverse directions. F means A transmits to B while R means B transmits to A. Situations where severe starvation occur are highlighted in gray. As seen from Table 3 while a power difference of greater than 3dB creates severe starvation scenarios in Schmidl-Cox, Skip-Correlation preserves carrier sensing symmetry.

8 Related Work

Carrier sensing has been extensively studied [16] and has received a lot of attention from the research community.

Improving performance through tuning carrier sensing. It is well-known that carrier sensing is limited by the fact that carrier is sensed at the sender whose channel measurements may be very different from that of the receiver. This could result in problems such as hidden terminal as well as missed transmission opportunities termed as exposed terminals. There has been a lot of work on improving Wi-Fi performance by tuning carrier sense threshold, along with transmit power, and/or transmission data rate, both for wireless LANs as well as wireless multihop networks [13, 15, 27, 28, 29, 30]. However, all these papers take a network-wide view where the parameters of every node can be tuned.

Carrier Sense Asymmetry. Starvation in Wi-Fi networks occurs due to variety of reasons such as flow-in-the-middle, hidden nodes, and sensing asymmetry [14]. In this paper, we focus on the carrier asymmetry issue due to transmit power differences which is known to cause starvation. To address starvation issues arising due to carrier sense asymmetry, authors in [18, 20, 23] propose solutions that jointly tune transmit power and carrier sense threshold of nodes in the network. Authors in [12] propose that nodes use lower transmit power and data rate for increased overall performance in dense and unmanaged wireless settings. In contrast, we present a solution that ensures carrier sense symmetry despite nodes choosing their transmit powers independently.

Coexistence with multiple transmit powers. Authors in [25] show that transmit power differences in white space networks can result in starvation of low power mobile nodes and propose a direct sequence spread spectrum-based solution to mitigate it. The Weeble system [22] also tackles the two power asymmetry problem in white space networks using a unique preamble for high power nodes that is different from the preamble for low-power nodes. As we show in this paper, the use of a unique preamble per power-level is expensive both in terms of preamble length as well as computational complexity. Instead, we address the problem of carrier sense asymmetry efficiently through the use of a single preamble while supporting a range of power levels.

9 Discussion

Are power differences having a major impact in Wi-Fi performance today? Wi-Fi performance issues are caused by numerous factors (e.g., interference, congestion, signal attenuation, power difference, etc.). Since power difference is only one among many causes for poor Wi-Fi performance, it is not clear how often it is the root cause of Wi-Fi performance problems. However, there are a few indications that power differences may be a significant problem. First, transmit power difference in Wi-Fi devices are common. For example, there are several devices available in the market whose transmit power ranges from 100mW (e.g. Samsung Galaxy S3 tx power is 111 mW) to 1W (e.g. Linksys EA6900 router, tx power 944 mW). Further, adapters are available for mobile laptops/PCs that allow 1W transmission [6]. Thus, there is both demand and supply of higher power Wi-Fi APs and devices in the market today. Second, this topic has received attention by the FCC. For example, FCC recently reached a settlement with TP-Link, a Wi-Fi AP manufacturer, in an investigation [5] where TP-Link was determined to have enabled a software-based setting for higher than allowed transmit power in some channels.

How much power difference can be supported efficiently? Skip-correlation is a general technique that can support arbitrary power differences (the 6dB limitation is only if backward compatibility to Wi-Fi is desired). However, efficiency can suffer if the range of power difference is large since the preamble needs to be appropriately elongated and long preambles may require larger Wi-Fi slots, thereby negatively impacting efficiency [19]. Given devices with transmit powers between 100 mW to 1000 mW are common in the market today, we believe that support for 10dB power difference can cater to a large number of scenarios. Based on our implementation of skip-correlator, the length of the preamble required to support 10x power difference with similar carrier sensing behavior as 100mW standard Wi-Fi is 400 samples (160+160 samples for skip correlation + 80 samples for AGC). A 400 sample preamble can fit in an existing 9μs Wi-Fi slot and will thus not reduce Wi-Fi efficiency.

Interactions with rate adaptation. Skip Correlation depends only on modifications to the PLCP Preamble. The SIGNAL field, which follows immediately after Preamble, encodes the rate in its first four bits. Since the preamble has already been received before the SIGNAL field, carrier sense is independent of the rate chosen. Thus, *skip-correlation has no impact on standard Wi-Fi rate adaptation and allows any rate adaptation algorithms to be used*. However, for systems that use rateless codes [21], it is not clear how best to incorporate (multi-power) carrier sensing.

10 Conclusion

We propose a novel carrier sensing solution, skip-correlation, to address the starvation of low-power devices given the increasing proliferation of high power devices in unlicensed spectrum.

11 Acknowledgments

We would like to thank our shepherd, Heather Zheng, and the anonymous reviewers of NSDI'17 for their comments towards improving the paper.

References

[1] Aironet 1570 Series Datasheet. http://www.cisco.com/c/dam/en/us/products/collateral/wireless/aironet-1570-series/datasheet-c78-732348.pdf.

[2] Apple Airport Express Specificaitons. http://www.apple.com/airport-express/specs/.

[3] Cisco Aironet router noise floor of -95/-93dBm. http://www.cisco.com/c/en/us/products/collateral/wireless/aironet-1830-series-access-points/datasheet-c78-735582.html.

[4] Dynamic Spectrum Alliance. http://www.dynamicspectrumalliance.org/.

[5] FCC Settlement with TP-Link. http://transition.fcc.gov/Daily_Releases/Daily_Business/2016/db0801/DOC-340564A1.pdf.

[6] High power Wi-Fi adapters. http://www.ebay.com/itm/LAPTOP-DESKTOP-WIFI-WLAN-ADAPTER-POWERFUL-SIGNAL-BOOSTER-DUAL-ANTENNA-MIMO-2T2R-/281506806802).

[7] HP Enterprise router noise floor of -97/-95dBm. http://www8.hp.com/h20195/v2/GetPDF.aspx/4AA5-1459ENW.pdf.

[8] RF Concepts - The Basics, 2012. http://community.arubanetworks.com/t5/Community-Tribal-Knowledge-Base/RF-Concepts-The-Basics/ta-p/25378.

[9] IEEE 802.11ac-2013, 2013. IEEE Standard for Information technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications – Amendment 4: Enhancements for Very High Throughput in Bands Below 6GHz.

[10] LTE Aggregation and Unlicensed Spectrum, 2015. http://www.4gamericas.org/files/1214/4648/2397/4G_Americas_LTE_Aggregation__Unlicensed_Spectrum_White_Paper_-_November_2015.pdf.

[11] The global public Wi-Fi network grows to 50 million worldwide hotspots, 2015. http://www.marketwired.com/press-release/the-global-public-wi-fi-network-grows-to-50-million-worldwide-wi-fi-hotspots-nasdaq-ipas-1984287.htm.

[12] A. Akella, G. Judd, S. Seshan, and P. Steenkiste. Self-management in chaotic wireless deployments. *Wireless Networks*, 13(6):737–755, 2007.

[13] J. Fuemmeler, N. H. Vaidya, and V. V. Veeravalli. Selecting transmit powers and carrier sense thresholds for CSMA protocols. *University of Illinois at Urbana-Champaign Technical Report*, 2004.

[14] C. Hua and R. Zheng. Starvation Modeling and Identification in Dense 802.11 Wireless Community Networks. In *IEEE Infocom*, 2008.

[15] T.-S. Kim, H. Lim, and J. C. Hou. Improving spatial reuse through tuning transmit power, carrier sense threshold, and data rate in multihop wireless networks. In *ACM MobiCom*, pages 366–377, 2006.

[16] L. Kleinrock and F. A. Tobagi. Packet switching in radio channels: Part I–carrier sense multiple-access modes and their throughput-delay characteristics. *IEEE Transactions on Communications*, 23(12):1400–1416, 1975.

[17] S. Krantz. *Handbook of Complex Variables*. Birkhauser, 1999.

[18] X. Liu, S. Seshan, and P. Steenkiste. Interference-aware transmission power control for dense wireless networks. In *In Proceedings of the Annual Conference of ITA*. Citeseer, 2007.

[19] E. Magistretti, K. K. Chintalapudi, B. Radunovic, and R. Ramjee. WiFi-Nano: reclaiming WiFi efficiency through 800 ns slots. In *ACM MobiCom*, pages 37–48, 2011.

[20] V. P. Mhatre, K. Papagiannaki, and F. Baccelli. Interference mitigation through power control in high density 802.11 WLANs. In *IEEE INFOCOM*, pages 535–543, 2007.

[21] J. Perry, P. A. Iannucci, K. E. Fleming, H. Balakrishnan, and D. Shah. Spinal codes. In *ACM SIGCOMM*, pages 49–60, 2012.

[22] B. Radunović, R. Chandra, and D. Gunawardena. Weeble: Enabling low-power nodes to coexist with high-power nodes in white space networks. In *ACM CoNeXT*, pages 205–216, 2012.

[23] M. Richart, J. Visca, and J. Baliosian. Self management of rate, power and carrier-sense threshold for interference mitigation in ieee 802.11 networks. In *IEEE CNSM*, pages 264–267, 2014.

[24] T. M. Schmidl and D. C. Cox. Robust frequency and timing synchronization for OFDM. *IEEE Transactions on Communications*, 45(12):1613–1621, 1997.

[25] S. Sur and X. Zhang. Bridging link power asymmetry in mobile whitespace networks. In *IEEE INFOCOM,*, pages 1176–1184, 2015.

[26] F. Tufvesson, O. Edfors, and M. Faulkner. Time and frequency synchronization for OFDM using PN-sequence preambles. In *IEEE VTC*, volume 4, pages 2203–2207, 1999.

[27] A. Vasan, R. Ramjee, and T. Woo. ECHOS-enhanced capacity 802.11 hotspots. In *IEEE INFOCOM*, volume 3, pages 1562–1572, 2005.

[28] Y. Yang, J. C. Hou, and L.-C. Kung. Modeling the effect of transmit power and physical carrier sense in multi-hop wireless networks. In *IEEE INFO-*

COM, pages 2331–2335, 2007.

[29] J. Zhu, X. Guo, S. Roy, and K. Papagiannaki. CSMA self-adaptation based on interference differentiation. In *IEEE Globecom*, pages 4946–4951, 2007.

[30] J. Zhu, X. Guo, L. L. Yang, and W. S. Conner. Leveraging spatial reuse in 802.11 mesh networks with enhanced physical carrier sensing. In *IEEE ICC*, volume 7, pages 4004–4011, 2004.

Appendix

A Proof of P1 and P2 Section 5.3

Proof of P1. The second piece of $\mathbf{S}_{P_k}^{Recv}$ starts at $\gamma\frac{P_N}{P_k}$ and $\mathbf{S}_{P_i}^{Xmit}$ ends at sample number $\gamma\frac{P_N}{P_i}$. When $P_i \geq P_k$, $\frac{P_N}{P_k} \geq \frac{P_N}{P_i}$, thus $\mathbf{S}_{P_i}^{Xmit} \cap \mathbf{S}_{P_k}^{Recv}$ comprises only the first piece of $\mathbf{S}_{P_k}^{Recv}$ *i.e.,* $\mathbf{S}[1,\gamma]$.

Proof of P2. Any preamble piece is always completely contained between the ends of $\mathbf{S}_{P_j}^{Xmit}$ and $\mathbf{S}_{P_{j+1}}^{Xmit}$. Thus, when $i < k$ there are $k-i$ preamble pieces starting at the end of $\mathbf{S}_{P_k}^{Xmit}$ until the end of $\mathbf{S}_{P_i}^{Xmit}$. The total number of samples $N(i,k)$ in $\mathbf{S}_{P_i}^{Xmit} \cap \mathbf{S}_{P_k}^{Recv}$ is given by,

$$N(i,k) = \gamma + \sum_{j=1}^{j=k-i} \lambda_{k,i} = \gamma + P_k\gamma\left(\frac{1}{P_i} - \frac{1}{P_k}\right) = \frac{P_k}{P_i}\gamma. \tag{8}$$

B Sensing Through Preamble Correlation

In this section we describe the preamble correlation process and provide the necessary background.

The Problem of Preamble Detection. Let $\mathbf{S}(n)$ be the preamble of length L samples that is transmitted. The received signal $\mathbf{S}_{recv}(n)$ when no preamble is transmitted and when a preamble is transmitted given by,

$$\mathbf{S}_{recv}(n) = \begin{array}{cc} \mathbf{N}(n) & \text{no Xmit} \\ \mathbf{R}(n) + \mathbf{N}(n) & \text{Xmit} \end{array} \tag{9a}$$

$$\mathbf{R}(n) = \mathbf{S}(n) * \mathbf{H}. \tag{9b}$$

In Eqns 9a,9b, $\mathbf{N}(n)$ is the receiver noise, $\mathbf{H}$ is the channel response that captures how the channel transforms the transmitted signal into $\mathbf{R}(n)$ due to effects such as multi-path fading and $*$ is the convolution operation. The goal of detection techniques is to reliably distinguish between the *no transmission* and *transmission* scenarios in Eqn 9a based on $\mathbf{S}_{recv}(n)$. While there exist several detection schemes, we now describe the two most popular schemes – the *matched filter* and *Schmidl-Cox* [24].

Matched Filter. In this scheme the receiver computes the normalized cross-correlation of the preamble with the received signal as,

$$C = \left|\sum_{i=1}^{i=L} \mathbf{S_{recv}^{norm}}(i)\mathbf{S^{norm}}(i)\right|^2 \tag{10}$$

In Eqn 10 $\mathbf{S^{norm}}$ and $\mathbf{S_{recv}^{norm}}$ are scaled versions of $\mathbf{S}$ and $\mathbf{S}_{recv}$ respectively such that its total energy is 1. If $C > C_{th}$, the channel is deemed busy and otherwise not.

Schmidl and Cox Detection. Here $\mathbf{S}(n)$ comprises two identical halves *i.e.,* $\mathbf{S}(i) = \mathbf{S}(i + \frac{L}{2})$. As the transmission passes through the channel, each of its halves are effected by the channel in exactly the same way. Consequently $\mathbf{R}(i) = \mathbf{R}(i + \frac{L}{2})$. The receiver correlates two consecutive $\frac{L}{2}$ length windows from the received signal as,

$$C = \left|\sum_{i=1}^{i=\frac{L}{2}} \mathbf{S}_{recv}(i)\mathbf{S}_{recv}(i + \frac{L}{2})\right|^2 \tag{11}$$

A high correlation ($C > C_{th}$) indicates that a preamble was transmitted and the channel is deemed busy.

Why Matched Filter is Typically Not Used for CS in Practice. The matched filter technique has two significant drawbacks. First, due to frequency selective fading in the channel, $R(n)$ is often very different from $S(n)$ in Eqn 9b. Consequently, correlating $R(n)$ with $S(n)$ typically yields a low correlation. Schmidl-Cox does not suffer from this problem since, both halves are affected by the channel identically and thus results in a high correlation. Second, as discussed in Section 6, the circuit complexity of implementing a matched filter can be two orders of magnitude greater than that of Schmidl-Cox. Thus, in the rest of this section we confine our discussion to Schmidl-Cox detection.

Probability Distributions of C in Schmidl-Cox. Suppose that the noise floor is σ_N^2 (variance of noise), the distribution of C when no preamble is transmitted is given by,

$$P(C|\text{no Xmit}, \sigma_N^2) = P\left(\left|\sum_{i=1}^{i=\frac{L}{2}} \mathbf{N}(i)\mathbf{N}(i + \frac{L}{2})\right|^2\right) \tag{12}$$

Invoking the central limit theorem, the sum of $\mathbf{N}(i)\mathbf{N}(i + \frac{L}{2})$ is a zero mean Gaussian with variance $\frac{L}{2}\sigma_N^4$. Thus, $P(C|\text{no Xmit}, \sigma_N^2)$is a χ_1^2 distribution with mean $\frac{L}{2}\sigma_N^4$.

Let the received preamble energy per sample (signal power) be $\mathbf{E}(\mathbf{R}(i)\mathbf{R}(i)) = \sigma_S^2$ ($\mathbf{E}$ denotes expectation). Further, typical preambles also have $\mathbf{E}(\mathbf{R}(i)) = 0$. Since $\mathbf{R}(i) = \mathbf{R}(i + \frac{L}{2})$, the distribution of C when a preamble is transmitted is given by,

$$P(C|\text{Xmit}, \sigma_N^2, \sigma S^2) = \tag{13}$$
$$P\left(\left|\sum_{i=1}^{i=\frac{L}{2}} \sigma_S^2 + 2\mathbf{R}(i)\mathbf{N}(i) + \mathbf{N}(i)\mathbf{N}(i + \frac{L}{2})\right|^2\right)$$

Invoking central limit theorem, the sum of $\mathbf{R}(i)\mathbf{N}(i)$ is a zero mean Gaussian with variance $\frac{L}{2}\sigma_N^2\sigma_S^2$. Thus, the distribution of $P(C|\text{ Xmit}, \sigma_N^2, \sigma S^2)$ is a non-central χ_1^2 distribution with mean at $\frac{L^2}{4}\sigma_S^4 + L\sigma_N^2\sigma_S^2 + \frac{L}{2}\sigma_N^4$.

Thus, the probability distribution of computed correlations in the presence and absence of a preamble are χ_1^2 distributions with means given by Eqns 14a, 14b respectively as,

$$\mu(C) = \begin{cases} LE_N^2/2 & \text{no preamble (14a)} \\ (LE_S)^2/4 + (LE_S)E_N + LE_N^2/4 & \text{preamble (14b)} \end{cases}$$

While we have only shown the above property for Schimdl-Cox, it holds for other correlation based techniques as well.

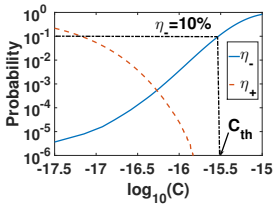


Figure 22: How threshold C_{th} is selected in Schmidl-Cox

Selection of $\mathbf{C_{th}}$. There can be two kinds of errors in preamble detection. False negatives occur when a preamble is transmitted but not detected while false positives occur when a preamble was not transmitted but erroneously detected. The false negative rate is given by,

$$\eta_- \left(C|\sigma_N^2, \sigma_S^2\right) = \int_{c=0}^{C} P(c|\text{Xmit}, \sigma_N^2, \sigma_S^2)dc \qquad (15)$$

and the false positive rate is given by,

$$\eta_+ \left(C|\sigma_N^2\right) = 1 - \int_{c=0}^{C} P(c|\text{no Xmit}, \sigma_N^2)dc \qquad (16)$$

The selection of $\mathbf{C_{th}}$ is based on the Wi-Fi specification of 10% false negative rate (90% detection rate) when the average received power is -82dBm [9]. As an example, Figure 22 depicts the curves for η_- and η_+ for a noise floor of -91dBm and L=8 samples ($0.4\mu s$). C_{th} is chosen for $\eta_- = 10\%$ (90% detection rate). Note that corresponding to this value of C_{th}, the false positive probability is well below 10^{-6}.

Effect of False Positives. A high false positive rate (η_+) leads a device to erroneously conclude that there is an ongoing transmission, resulting in an unnecessary backoff and loss of throughput. For a false detection rate of $\eta_=10^{-2}$, once every 100 samples ($5\mu s$), a preamble will be falsely detected. Since $5\mu s < 9\mu s$ (WiFi slot width) this means that the WiFi device will end up falsely detecting a preamble in almost every slot and will not be able to transmit any packets. Usually a false detection rate of $\eta_- < 10^{-3}$ is needed for reasonable performance. At 10^{-3}, once every 1000 samples or 6 Wi-Fi slots, a preamble is falsely detected. However, upon detecting an STS, the device then uses the Long Training Sequence (LTS) (Figure5) to tune its receiver parameters. If no LTS is detected then the receiver deems the channel free. The probability of falsely detecting a preamble in the LTS as well is now very low ($< 10^{-6}$). This means that the device will typically lose on average one WiFi slot for every packet transmission. Given that typical WiFi packet transmissions last several 100s of μs to few milliseconds, this results in a throughput drop of <2%.

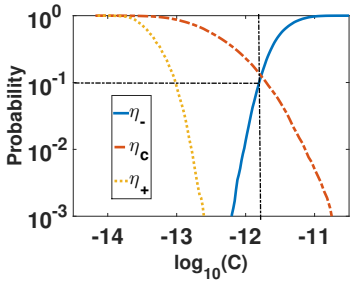


Figure 23: Performance of Orthogonal Preambles for 4 power levels and $L = 128$.

Approximating Detection Probability for Low False Rate Scenario. When the false positive rate is very low, the typical operating regime of receivers, $(LE_S)^2/4 >> (LE_S)E_N + LE_N^2/4$ and thus the mean reduces approximately to $(LE_S)^2/4$. Consequently, the detection probability is dependent on the product (LE_S).

C Performance of Orthogonal Preambles

In this section we consider the practical viability of the orthogonal preamble scheme proposed in Section 3 that uses a bank of matched filters. For this experiment we generated 4 different PN sequences of length 128 corresponding to each of the four power levels. In order to generate a PN sequence, we first generated a random vector comprising 128 complex numbers, each of which had a random phase between 0 to 360 degrees but had unit magnitude. Then we took an Inverse Fourier Transform of this random vector to create the PN sequence. This method ensured that equal energy was transmitted at various parts to frequency band to make it robust to frequency selective deep fades. A device transmitted this sequence and a receiver received these samples. We then found the normalized cross-correlation with each of the 4 PN sequences. The sequence that gave the maximum correlation was chosen and the power level was identified as corresponding to this sequence. This meant that there would be errors due to erroneously concluding an incorrect power level in addition to false positives. The larger the number power levels the worse is the performance and longer the length of preamble that is required. Figure 23 shows the false detection rates due to noise η_+, false preamble detections where the power level was incorrectly inferred η_c and the false negative rates η_- as a function of correlation values when the received power is -82dBm for $L = 128$. As seen from Figure 23 even though at $L = 128$ η_- is very small, the incorrect iden-

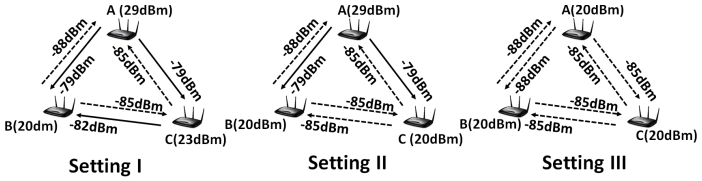


Figure 24: The setup for 3-node experiment

Setting-I						
Links	A-B	B-A	A-C	C-A	B-C	C-A
Schmidl-Cox	**100**	**0**	**97.8**	**12.9**	**24.2**	**99.8**
Skip	100	100	99.9	100	100	99.9
Setting-II						
Schmidl-Cox	**100**	**0**	**97.7**	**6.9**	24.1	33.7
Skip	100	100	97.4	99.6	39.7	41.7
Setting-III						
Schmidl-Cox	24.2	33.6	0	0	6.9	0
Skip	39.8	41.6	0.2	6.5	20.8	3.5

Table 4: Detection Probabilities for various links in 3 node experiment [%]

tification of power levels (η_c) becomes the dominating cause for errors. Table 5 shows the various values of η_c and η_+ for different values of L. While η_+ is acceptably low at $L = 128$, η_c is very high at 14%. This experiment shows why orthogonal preamble approach is not only extremely computationally inefficient (Section 6) but also performs poorly.

L	8	16	32	64	128
η_c	69.8	58.12	54.5	36.08	14.22
η_+	11.08	9.05	6.14	1.53	< 0.1

Table 5: False positive rates [%] for various lengths of orthogonal preambles

D CS Symmetry - Three Node Experiments

In this experiment we placed three WARP boards A, B and C at three different fixed locations and change their transmit powers to create three different scenarios - Setting I, Setting II and Setting III depicted in Figure 24. The transmit powers of the devices and their average received powers over the course of the experiment are also depicted in Figure 24. Similar to our experiment with two nodes, in each setting we run 5000 rounds where all devices transmit once each in a single round. We wait about 1 second between each round. The total duration of this experiment was about 5 hours and there was about 5-8dB variation in received signal strength during the course of the experiment. We use both Schmidl-Cox as well as Skip-Correlation to evaluate the detection probabilities, which are provided in Table 4. X-Y indicates A transmitting to Y.

Setting-I: This setting is similar in the example illustrated in Figure 12(b) where A can be sensed by both B and C while the reverse is not true. Also, B can sense C but not the other way round. As seen from Table 4, Schmidl-Cox experiences carrier sensing asymmetry in all three pairs as expected, however in Skip-Correlation this the detection probabilities are symmetric in either link directions.

Setting-II: In this setting we reduce C's power to 20dBm to remove the asymmetry between B and C. Thus, both B and C can sense A but not each other. As in Setting-I, A cannot sense both B and C. As seen from Table 4, Schmidl-Cox experiences shows carrier sensing asymmetry between A,B and A,C pairs while Skip-Correlation exhibits carrier sensing symmetry between all pairs.

Setting-III: In this setting we reduce the power of A to 20dBm to remove all link asymmetries. As seen from Table 4, both Schmidl-Cox and Skip-Correlation do not show any carrier sensing asymmetry as expected.

These results demonstrate that Skip-Correlation preserves carrier sensing symmetry under various scenarios.

FM Backscatter: Enabling Connected Cities and Smart Fabrics

Anran Wang[†]*, Vikram Iyer*[†]*, Vamsi Talla, Joshua R. Smith and Shyamnath Gollakota*
University of Washington
[†]Co-primary Student Authors

Abstract – This paper enables connectivity on everyday objects by transforming them into FM radio stations. To do this, we show for the first time that ambient FM radio signals can be used as a signal source for backscatter communication. Our design creates backscatter transmissions that can be decoded on any FM receiver including those in cars and smartphones. This enables us to achieve a previously infeasible capability: backscattering information to cars and smartphones in outdoor environments.

Our key innovation is a modulation technique that transforms backscatter, which is a multiplication operation on RF signals, into an addition operation on the audio signals output by FM receivers. This enables us to embed both digital data as well as arbitrary audio into ambient analog FM radio signals. We build prototype hardware of our design and successfully embed audio transmissions over ambient FM signals. Further, we achieve data rates of up to 3.2 kbps and ranges of 5–60 feet, while consuming as little as 11.07 μW of power. To demonstrate the potential of our design, we also fabricate our prototype on a cotton t-shirt by machine sewing patterns of a conductive thread to create a smart fabric that can transmit data to a smartphone. We also embed FM antennas into posters and billboards and show that they can communicate with FM receivers in cars and smartphones.

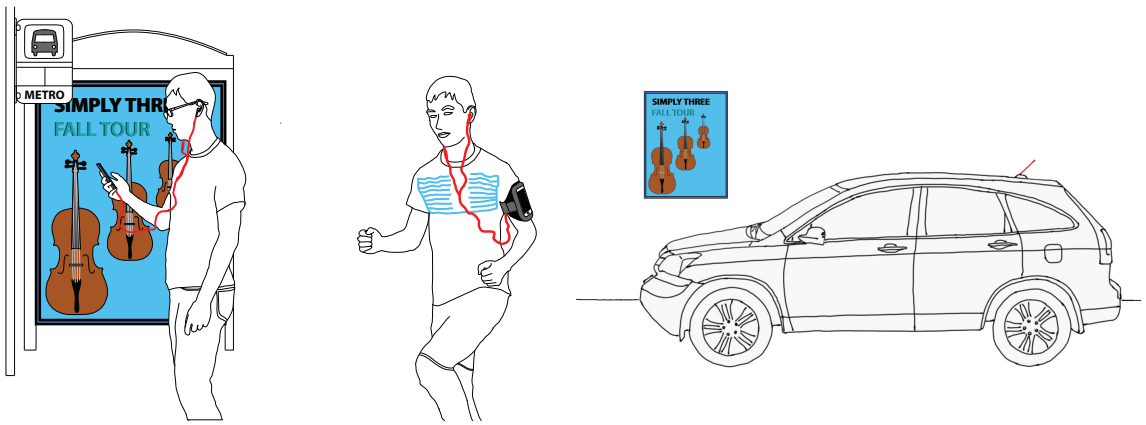
1 Introduction

This paper asks the following question: can we enable everyday objects in outdoor environments to communicate with cars and smartphones, without worrying about power? Such a capability can enable transformative visions such as connected cities and smart fabrics that promise to change the way we interact with objects around us. For instance, bus stop posters and street signs could broadcast digital content about local attractions or advertisements directly to a user's car or smartphone. Posters advertising local artists could broadcast clips of their music or links to purchase tickets for upcoming shows. A street sign could broadcast information about the name of an intersection, or when it is safe to cross a street to improve accessibility for the disabled. Such ubiquitous low-power connectivity would also enable smart fabric applications — smart clothing with sensors integrated into the fabric itself could monitor a runner's gait or vital signs and directly transmit the information to their phone.

While recent efforts on backscatter communication [40, 49, 38, 36] dramatically reduce the power requirements for wireless transmissions, they are unsuitable for outdoor environments. Specifically, existing approaches either use custom transmissions from RFID readers or backscatter ambient Wi-Fi [37] and TV transmissions [40, 42]. RFID-based approaches are expensive in outdoor environments given the cost of deploying the reader infrastructure. Likewise, while Wi-Fi backscatter [37] is useful indoors, it is unsuitable for outdoor environments. Finally, TV signals are available outdoors, but smartphones as well as most cars do not have TV receivers and hence cannot decode the backscattered signals.

Taking a step back, the requirements for our target applications are four-fold: 1) The ambient signals we hope to backscatter must be ubiquitous in outdoor environments, 2) devices such as smartphones and cars must have the receiver hardware to decode our target ambient signals, 3) it should be legal to backscatter in the desired frequencies without a license, which precludes cellular transmissions [31], and 4) in order to retrieve the backscattered data, we should ideally have a software-defined radio like capability to process the raw incoming signal without any additional hardware.

Our key contribution is the observation that FM radio satisfies the above constraints. Broadcast FM radio infrastructure already exists in cities around the world. These FM radio towers transmit at a high power of several hundred kilowatts [31] which provides an ambient signal source that can be used for backscatter communication. Additionally, FM radio receivers are included in the LTE and Wi-Fi chipsets of almost every smartphone [24]



(a) Smart posters and smartphones (b) Smart fabrics (c) Smart posters and cars

Figure 1: **Example applications enabled by backscattering FM radio signals.** (a) smart posters capable of broadcasting music to nearby users; (b) vital signs monitoring with sensors integrated into smart fabrics; (c) FM receivers in cars receiving music sent from nearby smart posters.

and have recently been activated on Android devices in the United States [23, 8]. Further, the FCC provides an exemption for low-power transmitters to operate on FM bands without requiring a license [30]. Finally, unlike commercial Wi-Fi, Bluetooth and cellular chipsets that provide only packet level access, FM radios provide access to the raw audio decoded by the receiver. These raw audio signals can be used in lieu of a software-defined radio to extract the backscattered data.

Building on this, we transform everyday objects into FM radio stations. Specifically, we design the first system that uses FM signals as an RF source for backscatter. We show that the resulting transmissions can be decoded on any FM receiver including those in cars and smartphones. Achieving this is challenging for two key reasons:

- Unlike software radios that give raw RF samples, FM receivers output only the demodulated audio. This complicates decoding since the backscatter operation is performed on the RF signals corresponding to the FM transmissions while an FM receiver outputs only the demodulated audio. Thus, the backscatter operation has to be designed to be compatible with the FM demodulator.
- FM stations broadcast audio that ranges from news channels with predominantly human speech, to music channels with a richer set of audio frequencies. In addition, they can broadcast either a single stream (mono mode) or two different audio streams (stereo mode) to play on the left and right speakers. Ideally, the backscatter modulation should operate efficiently with all these FM modes and audio characteristics.

To address these challenges, we leverage the structure of FM radio signals. At a high level, we introduce a modulation technique that transforms backscatter, which is a multiplication operation on RF signals, into an addition operation on the audio signal output by FM receivers (see §3.3). This allows us to embed audio and data information in the underlying FM audio signals. Specifically, we use backscatter to synthesize baseband transmissions that imitate the structure of FM signals, which makes it compatible with the FM demodulator. Building on this, we demonstrate three key capabilities.

- *Overlay backscatter.* We overlay arbitrary audio on ambient FM signals to create a composite audio signal that can be heard using any FM receiver, without any additional processing. We also design a modulation technique that overlays digital data which can be decoded on FM receivers with processing capabilities, e.g., smartphones.
- *Stereo backscatter.* A number of FM stations, while operating in the stereo mode, do not effectively utilize the stereo stream. We backscatter data and audio on these under-utilized stereo streams to achieve a low interference communication link. Taking it a step further, we can also trick FM receivers to decode mono FM signals in the stereo mode, by inserting the pilot signal that indicates a stereo transmission. This allows us to backscatter information in the interference-free stereo stream.
- *Cooperative backscatter.* Finally, we show that using cooperation between two smartphones from users in the vicinity of the backscattering objects, we can imitate a MIMO system that cancels out the audio in the underlying ambient FM transmission. This allows us to decode the backscatter transmissions without any interference.

To evaluate our design, we first perform a survey of the FM signal strengths in a major metropolitan city and identify unused spectrum in the FM band, which we then use in our backscatter system. We implement a prototype FM backscatter device using off-the-shelf components and use it to backscatter data and arbitrary audio directly to a Moto G1 smartphone and 2010 Honda CRV's FM receiver. We compare the trade offs for the three backscat-

ter techniques described above and achieve data rates of up to 3.2 kbps and ranges of 5–60 ft. Finally, we design and simulate an integrated circuit that backscatters audio signals, and show that it consumes only 11.07 μW.

To demonstrate the potential of our design, we build two proof-of-concept applications. We build and evaluate posters with different antenna topologies including dipoles and bowties fabricated using copper tape. We also fabricate our prototype on a cotton t-shirt by machine sewing patterns of a conductive thread and build a smart fabric that can transmit vital sign data to the phone, even when the user is walking or running.

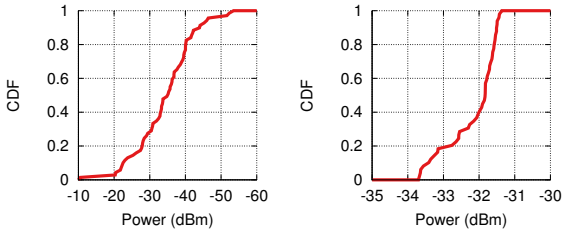
Contributions. We use backscatter to transform everyday objects into FM radio stations. In particular, we

- Introduce the first system to backscatter audio and data to cars and smartphones, outdoors. We present the first backscatter design that uses FM radio as its signal source and creates signals that are decodable on any FM receiver.
- Transform backscatter, which is a multiplication operation on RF signals, into an addition operation on the FM audio signals. Building on this, we present three key capabilities: overlay, stereo and cooperative backscatter.
- Build FM backscatter hardware and evaluate it in different scenarios. To show the potential of our design, we also build two proof-of-concept application prototypes for smart fabrics and smart bus stop billboards.

2 Application Requirements

Current radio technologies cannot satisfy the requirements for the connected city and smart fabric applications described above. Traditional radio transmitters are relatively expensive to deploy at scale and consume significant amounts of power; in contrast, we develop a backscatter system that consumes orders of magnitude less power and can be produced at lower cost. In the remainder of this section we outline the requirements of our target applications in detail and explain why FM backscatter is the only viable approach.

Connected cities. Connected city devices such as posters and signs must continuously broadcast information to anyone passing by. Considering most outdoor locations such as signposts and bus stops do not have dedicated connections to power infrastructure, these devices will either be battery powered, or harvest power from the environment. A practical communication solution should seek to maximize battery life so as to prevent recurring maintenance costs for these objects. A normal FM transmitter designed for battery powered applications [15] consumes 18.8 mA when transmitting. If the chip is continuously broadcasting, this system would last less than 12 hrs using a 225 mAh battery coin cell battery [12]. In practice the battery life would likely be much shorter considering the current draw of the FM chip is significantly higher than the rated 0.2 mA discharge current with which the battery was tested. In contrast, our backscatter system could continuously transmit for almost 3 years. Additionally, even at scale an FM radio chip costs over $4 [16] whereas a backscatter solution costs as little as a few cents [9].



(a) Across a major US city. (b) Across 24 hours of a day.

Figure 2: **Survey of FM radio signals.** (a) CDF of received power of FM radio signals across a major metropolitan city and (b) CDF of powers at a fixed location over a 24 hour duration.

Another potential alternative is Bluetooth Low Energy, such as the iBeacon standard that is designed for sending broadcast messages. In broadcast mode however, the BLE protocol is limited to sending short packets every 100 ms, and therefore not a viable solution for streaming audio. Additionally, while some cars have Bluetooth connectivity, their Bluetooth antennas are positioned inside to interact with mobile phones and other devices. An antenna inside the car would be shielded from our smart objects by the car's metal body, causing significant attenuation. In contrast, FM antennas are already positioned outside the car as this is where FM signals are strongest.

Smart fabrics. In addition to the same power constraints described above, smart fabrics require thin and flexible form factors. Materials such as conductive thread allow us to produce flexible FM antennas on textile substrates, which blend into cloth. However, commercially available flexible batteries have a limited discharge current. For example current flexible batteries on the market are limited to a 10 mA peak current [6] and cannot satisfy the requirements of FM or BLE radios when transmitting [15, 18]. While the reader may assume that the size of FM antennas limits smart fabric applications, we demonstrate that by using conductive thread we can take advantage of the whole surface area available on a garment to produce a flexible and seamlessly integrated antenna.

3 System Design

We use backscatter to encode audio and digital data on FM radio signals. In this section we begin by evaluating the availability and characteristics of FM radio broadcasts in urban areas. Next we provide background on FM radio

and how we can leverage its signal structure for backscatter communication. Finally, we describe our encoding mechanism that embeds digital data using backscatter.

3.1 Survey of FM Radio Signals

Public and commercial FM radio broadcasts are a standard in urban centers around the world and provide a source of ambient RF signals in these environments. In order to cover a wide area and allow for operation despite complex multipath from structures and terrains, FM radio stations broadcast relatively high power signals. In the United States, FM radio stations produce an effective radiated power of up to 100 kW [31].

In this section, we survey the signal strength of FM radio transmissions across Seattle, WA. We drive through the city and measure the power of ambient FM radio signals using a software defined radio (SDR, USRP E310) connected to a quarter-wavelength monopole antenna (Diamond SRH789). Since there are multiple FM radio stations in most US cities, we record signals across the full 88–108 MHz FM spectrum and identify the FM station with the maximum power at each measurement location. We calibrate the raw SDR values to obtain power measurements in dBm using reference power measurements performed with a spectrum analyzer (Tektronix MDO4054B-3). We divide the surveyed area into 0.8 mi× 0.8 mi grid squares and determine the median power in each for a total of 69 measurements.

Fig. 2a shows the CDF of the measurements across the city. The plot shows that the FM signal strength varies between -10 and -55 dBm. We also note that the median signal strength across all the locations is -35.15 dBm, which is surprisingly high, despite measurements being performed in a dense urban environment with tall buildings and across elevation differences of 450 ft.[1] We note that, prior work on ambient backscatter reflecting TV signals, requires the strength of TV signals to be at least -10 to -20 dBm. In contrast, our goal is to decode the backscattered signals on FM receivers which have a sensitivity around -100 dBm [14, 1]. Thus, FM signals are promising as a signal source for backscatter.

Next, we place the same SDR setup in a fixed outdoor location over the course of 24 hours. We record 100,000 samples over the whole FM radio band and compute the average power of the FM station with the highest power once every minute. Fig. 2b shows that the power varies with a standard deviation of 0.7 dBm, which shows that the received power is roughly constant across time. These measurements show that FM signals can provide a reliable signal source for backscatter.

[1]For comparison, at -50 dBm Wi-Fi can support hundreds of Mbps.

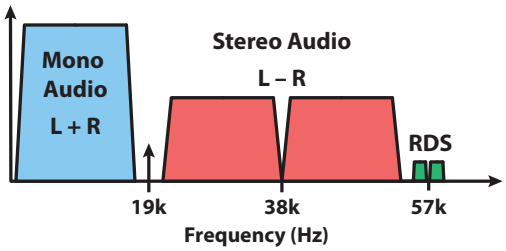


Figure 3: **Structure of FM Radio Transmissions.** Baseband audio signals transmitted in stereo FM broadcasts.

3.2 Structure of FM Radio Transmissions

We leverage the structure of FM signals to embed audio and data. In this section, we outline the background about FM radio necessary to understand our design.

Audio Baseband Encoding. Fig. 3 illustrates the baseband audio signal transmitted from a typical FM station. The primary component is a mono audio stream, which is an amplitude modulated audio signal between 30 Hz and 15 kHz. With the advent of stereo speakers with separate left and right audio channels, FM stations incorporated a stereo stream transmitted in the 23 to 53 kHz range. To maintain backward compatibility, the mono audio stream encodes the sum of the left and right audio channels (L+R) and the stereo stream encodes their difference (L-R). Mono receivers only decode the mono stream, while stereo receivers process the mono and stereo streams to separate the left (L) and right (R) audio channels. The FM transmitter uses a 19 kHz pilot tone to convey the presence of a stereo stream. Specifically, in the absence of the pilot signal, a stereo receiver would decode the incoming transmission in the mono mode and uses the stereo mode only in the presence of the pilot signal. In addition to the mono and stereo audio streams, the transmitter can also encode the radio broadcast data system (RDS) messages that include program information, time and other data sent at between 56 and 58 kHz.

RF Encoding. As the name suggests, FM radio uses changes in frequency to encode data. Unlike packet based radio systems such as Bluetooth and Wi-Fi, analog FM radio transmissions are continuous in nature. An FM radio station can operate on one of the 100 FM channels between 88.1 to 108.1 MHz, each separated by 200 kHz. Specifically, FM radio signals are broadcast at the carrier frequency f_c in one of these FM bands, and information at each time instant is encoded by a deviation from f_c.

Mathematically, an FM transmission can be written as,

$$\text{FM}_{RF}(t) = \cos\left(2\pi f_c t + 2\pi\Delta f \int_0^t \text{FM}_{audio}(\tau)d\tau\right) \quad (1)$$

In the above equation, f_c is the carrier frequency of the RF transmission and $\text{FM}_{audio}(\tau)$ is the baseband audio

signal shown in Fig. 3. If we normalize $\text{FM}_{audio}(\tau)$ to be between -1 to 1, then Δf is the maximum deviation in frequency from the carrier frequency, f_c.

Using Carson's rule [28] we can approximate the bandwidth of the above signal as $2(\Delta f + \max(f_{audio}))$, where $\max(f_{audio})$ is the maximum frequency in the baseband audio signal. For FM radio stations broadcasting mono, stereo, and RDS up to 58 kHz, using a maximum Δf of 75 kHz [31] results in a bandwidth of 266 kHz.

FM Decoding. The FM receiver first demodulates the incoming RF signal to baseband to obtain $\text{FM}_{BB} = \cos\left(2\pi\Delta f \int_0^t \text{FM}_{audio}(\tau)d\tau\right)$. In order to extract the encoded audio information, the receiver performs a derivative operation. Specifically, the derivative $\frac{d}{dt}\text{FM}_{BB}(t)$ converts the frequency changes to amplitude variations:

$$-2\pi\Delta f \text{FM}_{audio}(t)\sin\left(2\pi\Delta f \int_0^t \text{FM}_{audio}(\tau)d\tau\right)$$

Dividing this by the phase shifted baseband FM signal, $\text{FM}_{BB}\left(t+\frac{\pi}{2}\right)$ recovers the original audio signal, $2\pi\Delta f \text{FM}_{audio}(t)$. Note that the amplitude of the decoded baseband audio signal is scaled by the frequency deviation Δf; larger frequency deviations result in a louder audio signal. In our backscatter prototype, we set this parameter to the maximum allowable value. We also note that while the above description is convenient for understanding, in practice FM receiver circuits implement these decoding steps using phase-locked loop circuits.

3.3 Backscattering FM Radio

Backscattering FM radio transmissions and decoding them on mobile devices is challenging because FM receivers only output the decoded audio, while the backscatter operation is performed on the RF signals. To address this, we show that by exploiting the FM signal structure, we can transform backscatter, which performs a multiplication operation in the RF domain, into an addition operation in the audio domain.

To understand this, let us first look at the backscatter operation. Say we have a signal source that transmits a single tone signal, $\cos(2\pi f_c t)$ at a center frequency, f_c. If the backscatter switch is controlled with the baseband signal $B(t)$, it generates the signal $B(t)\cos(2\pi f_c t)$ on the wireless medium. Thus, the backscatter operation performs multiplication in the RF domain. So, when we backscatter ambient FM radio signals, $\text{FM}_{RF}(t)$, we generate the backscattered RF signal, $B(t)\times\text{FM}_{RF}(t)$.

Say we pick $B(t)$ as follows:

$$B(t) = \cos\left(2\pi f_{back} t + 2\pi\Delta f \int_0^t \text{FM}_{back}(\tau)d\tau\right) \quad (2)$$

The above signal has the same structure as the FM radio transmissions in Eq. 1, with the exception that it is centered at f_{back} and uses the audio signal $\text{FM}_{back}(\tau)$. The backscattered RF signal $B(t)\times\text{FM}_{RF}(t)$ then becomes:

$$\cos\left(2\pi f_{back} t + 2\pi\Delta f \int_0^t FM_{back}(\tau)d\tau\right)\times$$
$$\cos\left(2\pi f_c t + 2\pi\Delta f \int_0^t FM_{audio}(\tau)d\tau\right)$$

The above expression is a product of two cosines, and allows us to apply the following trigonometric identity: $2\cos(A)\cos(B) = \cos(A+B) + \cos(A-B)$. Focusing on just the $\cos(A+B)$ term[2] yields:

$$\cos\left(2\pi\left[(f_c + f_{back})t + \Delta f \int_0^t \text{FM}_{audio}(\tau) + \text{FM}_{back}(\tau)d\tau\right]\right)$$

The key observation is that the above expression is of the same form as an FM radio signal centered at a frequency of $f_c + f_{back}$ with the baseband audio signal $\text{FM}_{audio}(\tau) + \text{FM}_{back}(\tau)$. Said differently, an FM radio tuned to the frequency $f_c + f_{back}$, will output the audio signal $\text{FM}_{audio}(t) + \text{FM}_{back}(t)$. Thus, by picking the appropriate backscatter signal we can use the multiplicative nature of backscatter in the RF domain to produce an addition operation in the received audio signals. We call this *overlay backscatter* because the backscattered information is overlaid on top of existing signals and has the same structure as the underlying data.

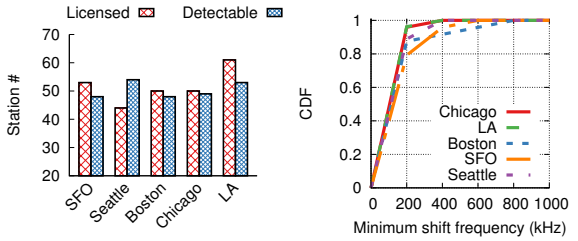
Next, we describe how we pick the parameters in Eq. 2.

1) How do we pick f_{back}? We pick f_{back} such that $f_c + f_{back}$ lies at the center of an FM channel. This allows any FM receiver to tune to the corresponding channel and receive the backscatter-generated signals. In addition, f_{back} should be picked such that the resulting FM channel is unoccupied. We note that due to the bandwidth of FM transmissions, geographically close transmitters are often not assigned to adjacent FM channels [31]. This, along with changes to station ownership and licensing policies over the years, has left various FM channels empty.

Fig. 4a shows the FM band occupancy in five different US cities. We show both the number of licensed stations in a city as well as the number of stations detected in a particular zip code in that city based on public resources available online [13, 7]. This figure shows that in some cities the number of detectable stations is less than the number of assigned channels, as some licensed stations may no longer be operational. Further, in cities like Seattle, there are more detectable bands than assigned licenses as transmissions from neighboring cities may be detected. The key point however is that a large fraction of 100 FM channels are unoccupied and can be used for backscatter.

Finally, to compute the f_{back} required in practice, we measure the frequency separation between each licensed

[2] The $\cos(A-B)$ term can be removed using single-sideband modulation as described in [36].



(a) (b)

Figure 4: **Usage of FM channels in US cities.** (a) shows both the number of licensed and detectable channels of the 100 FM bands and (b) shows the minimum frequency difference between licensed FM stations and the closest unoccupied FM band.

FM station and the nearest channel without a licensed station based on data from [13]. Fig. 4b shows the CDF of the minimum f_{back} across five major US cities. The plot shows that the median frequency shift required is 200 kHz and is less than 800 kHz in the worse case situation. We note that the optimal value of f_{back} will vary by location and should be chosen such that the backscatter transmission is sent at the frequency with the lowest power ambient FM signal. This is because while FM receivers often have very good sensitivities, in practice the noise floor may instead be limited by power leaked from an adjacent channel.

2) How do we pick $FM_{back}(\tau)$? This depends on whether we want to overlay audio or data on the ambient FM signals. Specifically, to overlay audio we set $\text{FM}_{back}(\tau)$ to follow the structure of the audio baseband signal shown in Fig. 3. To send data, we instead generate audio signals using the modulation techniques described in §3.4.

3) How do we use backscatter to generate Eq. 2? At a high level, Eq. 2 is a cosine signal with a time-varying frequency. Thus, if we can generate cosine signals at different frequencies using backscatter, we can create the required backscatter signal. To do this, we approximate the cosine signal with a square wave alternating between +1 and -1. These two discrete values can be created on the backscatter device by modulating the radar cross-section of an antenna to either reflect or absorb the ambient signal at the backscatter device. By changing the frequency of the resulting square wave, we can approximate a cosine signal with the desired time-varying frequencies.

3.3.1 FM Backscatter capabilities

The above description focuses on *overlay backscatter* where the backscattered audio data is simply overlaid on top of the ambient FM signals. In this section, we describe two additional backscatter techniques.

Stereo Backscatter. We consider two scenarios. 1) A mono radio station that does not transmit a stereo stream, and 2) A stereo station that broadcast news information.

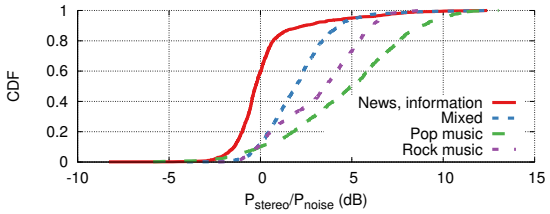


Figure 5: CDF illustrating the signal power broadcast in the stereo band (L-R) of various local FM stations.

1) Mono to stereo backscatter. While many commercial FM radio stations broadcast stereo audio, some stations only broadcast a mono audio stream. In this case, all the frequencies corresponding to the stereo stream (15-58 kHz in Fig. 3) are unoccupied. Thus, they can be used to backscatter audio or data without interference from the audio signals in the ambient FM transmissions. Utilizing these frequencies however presents two technical challenges. First, the FM receiver must be in the stereo mode to decode the stereo stream, and second FM receivers do not provide the stereo stream but instead only output the left and right audio channels (L and R).

To address the first challenge, we note that FM uses the 19 kHz pilot signal shown in Fig. 3 to indicate the presence of a stereo stream. Thus, in addition to backscattering data, we also backscatter a 19 kHz pilot signal. Specifically, our backscatter signal $B(t)$ is:

$$\cos\left(2\pi f_{back}t + 2\pi\Delta f \int_0^t 0.9\text{FM}_{back}^{stereo}(\tau) + 0.1\cos(2\pi 19k\tau)d\tau\right)$$

Inserting this 19 kHz pilot tone indicates that the receiver should decode the full stereo signal which includes our backscattered audio $\text{FM}_{back}^{stereo}$.

To address the second challenge, we note that FM receivers do not output the stereo stream (L-R) but instead output the left and right audio streams. To recover our stereo backscatter signal, all we have to do is compute the difference between these left (L) and right (R) audio streams. This allows us to send data/audio in the unoccupied stereo stream of a mono FM transmission.

2) Stereo backscatter on news stations. While many FM stations transmit in the stereo mode, i.e. with the 19 kHz pilot tone in Fig. 3, in the case of news and talk radio stations, the energy in the stereo stream is often low. This is because the same human speech signal is played on both the left and right speakers. We verify this empirically by measuring the stereo signal from four different radio stations. We capture the audio signals from these stations for a duration of 24 hrs and compute the average power in the stereo stream and compare it with the average power in 16-18 kHz, which are the empty frequencies in Fig. 3.

Figure 5 shows the CDF of the computed ratios for the

four FM stations. These plots confirm that in the case of news and talk radio stations, the stereo channel has very low energy. Based on this observation, we can backscatter data/audio in the stereo stream with significantly less interference from the underlying FM signals. However, since the underlying stereo FM signals already have the 19 kHz pilot signal, we do not backscatter the pilot tone.

Cooperative backscatter. Consider a scenario where two users are in the vicinity of a backscattering object, e.g., an advertisement at a bus stop. The phones can share the received FM audio signals through either Wi-Fi direct or Bluetooth and create a MIMO system that can be used to cancel the ambient FM signal and decode the backscattered signal. Specifically, we set the phones to two different FM bands: the original band of the ambient FM signals (f_c) and the FM band of the backscattered signals ($f_c + f_{back}$). The audio signals received on the two phones can then be written as,

$$S_{phone1} = FM_{audio}(t)$$
$$S_{phone2} = FM_{audio}(t) + FM_{back}(t)$$

Here we have two equations in two unknowns, $\text{FM}_{audio}(t)$ and $\text{FM}_{back}(t)$, which we can solve to decode the backscattered audio signal $\text{FM}_{back}(t)$. In practice however we need to address two issues: 1) The FM receivers on the two smartphones are not time synchronized, and 2) On the second phone, hardware gain control alters the amplitude of $\text{FM}_{audio}(t)$ in the presence of $\text{FM}_{back}(t)$.

To address the first issue, we resample the signals on the two phones, in software, by a factor of ten. We then perform cross-correlation between the two resampled signals to achieve time synchronization between the two FM receivers. To address the second issue, we transmit a low power pilot tone at 13 kHz as a preamble that we use to estimate the amplitude. Specifically, we compare the amplitude of this pilot tone during the preamble with the same pilot sent during the audio/data transmission. We scale the power of the received signal by the ratio of the two amplitude values. This allows us to calibrate the amplitude of the backscattered signal. We then subtract these two signals to get the backscatter signal, $\text{FM}_{back}(t)$.

3.4 Data Encoding with Backscatter

We encode data using the audio frequencies we can transmit using backscatter. The key challenge is to achieve high data rates without a complex modulation scheme to achieve a low power design. High data rate techniques like OFDM have high computational complexity (performing FFTs) as well as have high peak-to-average ratio, which either clips the high amplitude samples, or scales down the signal and as a result limits the communication ranges. Instead we use a form of FSK modulation in combination with a computationally simple frequency division multiplexing algorithm, as described below.

Data Encoding process. At a high level the overall data rate of our system depends on both the symbol rate and the number of bits encoded per symbol. We modify these two parameters to achieve three different data rates. We present a low rate scheme for low SNR scenarios as well as a higher rate scheme for scenarios with good SNR.

100 bps. We use a simple binary frequency shift keying scheme (2-FSK) where the zero and one bits are represented by the two frequencies, 8 and 12 kHz. Note that both these frequencies are above most human speech frequencies to reduce interference in the case of news and talk radio programs. We use a symbol rate of 100 symbols per second, giving us a bit rate of 100 bps using 2-FSK. We implement a non-coherent FSK receiver which compares the received power on the two frequencies and output the frequency that has the higher power. This eliminates the need for phase and amplitude estimation and makes the design resilient to channel changes.

1.6 kbps and 3.2 kbps. To achieve high bit rates, we use a combination of 4-FSK and frequency division multiplexing. Specifically, we use sixteen frequencies between 800 Hz and 12.8 kHz and group them into four consecutive sets. Within each of these sets, we use 4-FSK to transmit two bits. Given that there are a total of four sets, we transmit eight bits per symbol. We note that, within each symbol, there are only four frequencies transmitted amongst the designated 16 to reduce the transmitter complexity. We choose symbol rates of 200 and 400 symbols per second allowing us to achieve data rates of 1.6 and 3.2 kbps. We note that our experiments showed that the BER performance degrades significantly when the symbol rates are above 400 symbols per second. Given this limitation, 3.2 kbps is the maximum data rate we achieve which is sufficient for our applications.

Maximal-ratio combining. We consider the original audio from the ambient FM signal to be noise, which we assume is not correlated over time; therefore we can use maximal-ratio combining (MRC) [46] to reduce the bit-error rates. Specifically, we backscatter our data N times and record the raw signals for each transmission. Our receiver then uses the sum of these raw signals in order to decode the data. Because the noise (*i.e.,* the original audio signal) of each transmission are not correlated, the SNR of the sum is therefore up to N times that of a single transmission.

4 Implementation

We build a hardware prototype with off-the shelf components, which we use in all our experiments and to build proof of concept prototypes for our applications. We then

design an integrated circuit based FM backscatter system and simulate its power consumption.

Off-the-shelf design. We use the NI myDAQ as our baseband processor, which outputs an analog audio signal from a file. For our FM modulator, we use the Tektronix 3252 arbitrary waveform generator (AWG) which has a built-in FM modulation function. The AWG can easily operate up to 10s of MHz and can generate an FM modulated square wave as a function of an input signal. Interfacing the NI myDAQ with the AWG gives us the flexibility of using the same setup to evaluate audio and data modulation for both mono and stereo scenarios. We connect the output of the AWG to our RF front end, which consists of the ADG902 RF switch. We design the switch to toggle the antenna between an open and short impedance state.

IC Design. In order to realize smart fabric and posters with FM backscatter, we translate our design into an integrated circuit to minimize both size and cost, and scale to large numbers. We implement the FM backscatter design in the TSMC 65 nm LP CMOS process. A detailed description of the IC architecture is presented below.

Baseband processor. The baseband data/audio is generated in the digital domain using a digital state machine. We write Verilog code for the baseband and translate it into a transistor level implementation using the Synopsis Design Compile tool. Our implementation of a system transmitting mono audio consumes 1 μW.

FM modulator. The FM modulator is based on an inductor and capacitor (LC) tank oscillator with an NMOS and PMOS cross-coupled transistor topology. We leverage the fact that the capacitors can be easily switched in and out and design a digitally controlled oscillator to modulate the oscillator frequency. We connect a bank of 8 binary weighted capacitors in parallel to an off-chip 1.8 mH inductor and control the digital capacitor bank from the output of the baseband processor to generate a FM modulated output. We simulate the circuit using Cadence Spectre [5]. Our 600 kHz oscillator with a frequency deviation of 75 kHz consumes 9.94 μW.

Backscatter switch. We implement the backscatter switch using an NMOS transistor connected between the antenna and ground terminals. The square wave output of the FM modulator drives the NMOS switch ON and OFF, which toggles the antenna between open and short impedance states to backscatter FM modulated signals. We simulate the switch in Cadence to show it consumes 0.13 μW of power while operating at 600 kHz. Thus, the total power consumption of our FM backscatter system is 11.07 μW.

5 Evaluation

We evaluate various aspects of our backscatter system.

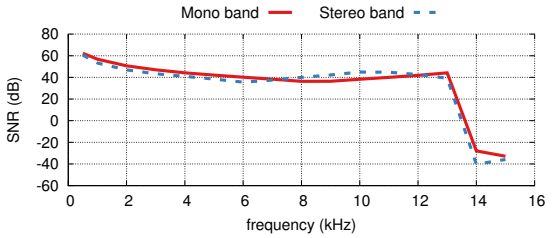


Figure 6: **SNR v/s frequencies.** Received SNR of different frequencies using Moto G1.

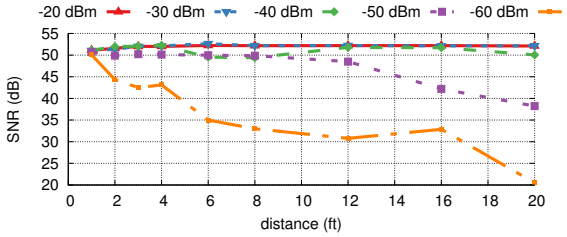


Figure 7: **SNR v/s power and distance.** Received SNR of different receiving powers and distances between the poster antenna and the receiver.

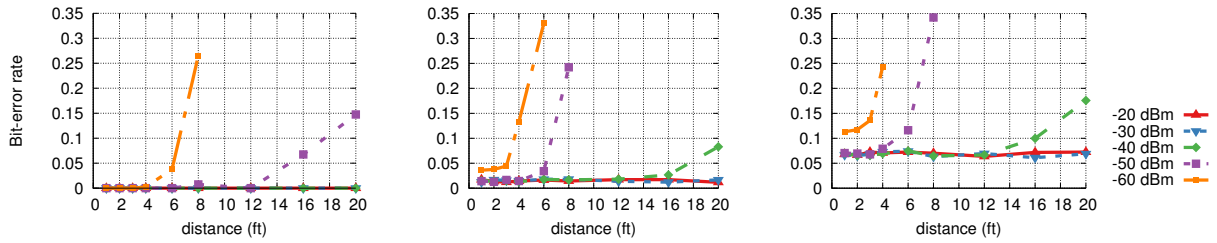
5.1 Micro-benchmarks

We first measure the frequency response of FM receivers to understand how they affect audio signals backscattered at different frequencies. We then benchmark the performance of our backscatter system versus distance.

Characterizing frequency response. We characterize the frequency response of our whole system by backscattering audio signals at different frequencies and analyzing the received audio signals. In order to benchmark our system without interfering background audio, we simulate an FM station transmitting no audio information (e.g. $\text{FM}_{audio} = 0$, which is a single tone signal at the channel center frequency f_c) using an SDR (USRP N210). Specifically, we set the USRP to transmit at 91.5 MHz. We then connect our prototype backscatter switch to the poster form factor antenna described in §6.1, which reflects the incoming transmissions and shifts them to an FM channel 600 kHz away by setting f_{back} in Eq. 2 to 600 kHz. We set the backscatter audio FM_{back} to single tone audio signals at frequencies between 500 Hz and 15 kHz.

We use a Moto G1 smartphone to receive these transmissions with a pair of Sennheiser MM30i headphones as its antenna. We use an FM radio app developed by Motorola [11] to decode the audio signals and save them in an AAC format. We perform these experiments in a shielded indoor environment to prevent our generated signals from interfering with existing FM stations.

We separate the backscatter antenna and FM receiver by 4 ft and place them equidistant from the FM transmitter. We then set the power at the FM transmitter such that the received power at the backscatter antenna is -20 dBm. We record the audio signal received on the phone and



(a) BFSK @ 100bps (b) FDM-4FSK @ 1.6kbps (c) FDM-4FSK @ 3.2kbps

Figure 8: **BER w/ overlay backscatter.** Bit-error rates using overlay backscatter *w.r.t.* different receiving powers and distances.

compute SNR by comparing the power at the frequency corresponding to the transmitted tone and the average power of the other audio frequencies. For instance, if the transmitter sends a single tone at 5 kHz, then we compute the ratio $\frac{P_{5\ kHz}}{\sum_f P_f - P_{5\ kHz}}$. We repeat the same experiment by transmitting the same 500 Hz to 15 kHz tones only in the L-R stereo band to determine whether the stereo receiver affects these signals differently.

Fig. 6 shows the frequency response for both the mono and stereo streams. The figure shows that the FM receiver has a good response below 13 kHz, after which there is a sharp drop in SNR. While this cut-off frequency is dependent on the FM receiver, recording app, and compression method used to store the audio, these plots demonstrate we can at the least utilize the whole 500 Hz to 13 kHz audio band for our backscatter transmissions.

Characterizing range versus distance. In the above set of experiments we fix the distance between the backscatter device and the FM transmitter as well as the power of the FM transmitter. Next, we vary these parameters to understand how the backscatter range changes as a function of the FM radio signal strength. To do this we measure how the SNR changes as a function of the distance between the backscatter device and the FM receiver at five different power levels from -20 dbm to -60 dbm. We set our prototype to backscatter a 1 kHz audio tone using the same setup described above. We then increase the distance between our backscatter device and the receiving phone while maintaining an equal distance from each device to the transmitter.

Fig. 7 shows the results. We can see that the backscatter device can reach 20 ft when the power of the FM transmitter is -30 dBm at the backscatter device. At a -50 dbm power level, the power in the backscattered signal is still reasonably high at close distances. This is because of the good sensitivity of the FM radio receivers in contrast to the TV based approaches explored in prior work such as ambient backscatter [40].

5.2 BER performance

Next, we evaluate how well our design can be used to transmit data using FM backscatter. We evaluate the three different bit rates of 100 bps, 1.6 and 3.2 kbps described in §3.4. Our goal in this evaluation is to understand the BER performance of these three different bit rates at different power levels for the ambient RF signals. Since it is difficult to control the power levels of the ambient RF signals in outdoor environments, we create an FM radio station by using a USRP to retransmit audio signals recorded from local FM radio stations. Specifically, we capture 8 s audio clips from four local FM stations broadcasting different content (news, mixed, pop music, rock music) and retransmit this audio in a loop at 91.5 MHz using our USRP setup. This setup allows us to test the effect of different types of background audio in an environment with precisely controllable power settings.

We again set f_{back} for our prototype to be 600 kHz, and set FM_{back} to continuous 8 s data transmissions at each of the bit rates described above. We vary the distance between the backscatter device and the smartphone and adjust the FM transmitter power to control the power received at the backscatter device.

We evaluate the BER with overlay backscatter where the data is embedded in the mono stream on top of the background audio signals transmitted by the USRP. Fig. 8 shows the BER results with the overlaid data as a function of distance between the FM receiver and the backscatter device for the three different bit rates. We also show the results for different power levels at the backscatter device. These plots show that:

- At a bit rate of 100 bps, the BER is nearly zero up to distances of 6 feet across all power levels between -20 and -60 dBm. Further, for power levels greater than -60 dBm, the range increases to over 12 feet. This shows that across all the locations in the survey in §3.1, we can transmit data to the smartphone using FM backscatter.
- Increasing bit rate reduces range. However, at power levels greater than -40 dBm, for both 1.6 and 3.2 kbps, the BERs are low at distances as high as 16 feet. Further, at 1.6 kbps, the BERs are still low up to 3 and 6 feet at -60 and -50 dBm respectively.

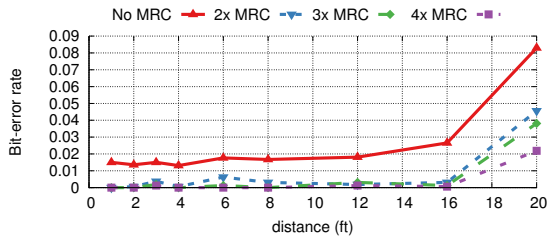


Figure 9: **BER w/ MRC.** Bit-error rates using overlay backscatter with MRC technique.

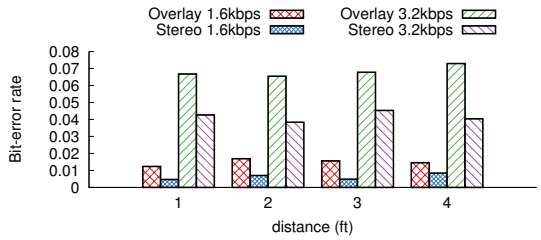


Figure 10: **BER w/ stereo backscatter.** Bit-error rates comparison using overlay backscatter and stereo backscatter.

• We can use the MRC combining technique described in §3.4 to further reduce the BER at the receiver. Fig. 9 shows the BER when performing MRC at a bit rate of 1.6 kbps and power level of -40 dBm. The figure shows the results for combining 2 to 4 consecutive transmissions. The results show that combining across two transmissions is sufficient to significantly reduce BER.

Finally, while MRC reduces the BER it also decreases the effective throughput at the receiver. To prevent this we can use stereo backscatter where the data is encoded in the stereo stream. We repeat a limited set of the above experiments to verify this. Specifically, we set the power received at the backscatter device to -30 dBm and set it to transmit data at bit rates of 1.6 kbps and 3.2 kbps respectively. We use the algorithm described in §3.3 to decode the backscatter information at the FM receiver. Fig. 10 shows the BER achieved using stereo backscatter. For comparison, we also plot the results for overlay backscatter. The plots show that stereo backscatter significantly decreases interference and therefore improves BER at the receiver. While stereo backscatter clearly improves performance, we note that it requires a higher power to detect the 19 kHz pilot signal and can therefore only be used in scenarios with strong ambient FM signals.

5.3 Audio Performance

Beyond data, our design also enables us to send arbitrary audio signals using FM backscatter. In this section, we evaluate the performance of FM audio backscatter. As before we set the FM transmitter to send four, 8 s samples of sound recorded from local radio stations. To evaluate the quality of the resulting audio signals, we use perceptual evaluation of speech quality (PESQ) which is a common

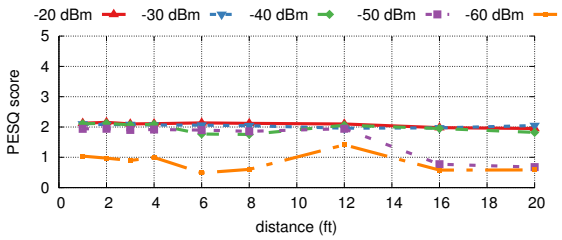


Figure 11: **PESQ w/ overlay backscatter.** PESQ scores of speech received using overlay backscatter *w.r.t.* different receiving powers and distances. In overlay backscatter, we have a background audio from the ambient FM signals and so what we hear is a composite signal. This sounds good at a PESQ value of two.

metric used to measure the quality of audio in telephony systems [35]. PESQ outputs a perception score between 0 and 5, where 5 is excellent quality. We evaluate this metric with overlay, stereo and collaborative backscatter.

Audio with overlay backscatter. In the case of overlay backscatter, we have two different audio signals: one from the backscatter device and the second from the underlying FM signals. We compute the PESQ metric for the backscattered audio information and regard the background FM signal as noise. We repeat the same experiments as before where we change the distance between the backscatter device and the Moto G1 smartphone at different power levels of FM signals. We repeat the experiments ten times for each parameter configuration and plot the results in Fig. 11. The plots shows that the PESQ is consistently close to 2 for all power numbers between -20 and -40 dBm at distance of up to 20 feet. We see similar performance at -50 dBm up to 12 feet. Unlike data, audio backscatter requires a higher power of -50 dBm to operate since one can perform modulation and coding to decode bits at a lower data rate at down to -60 dBm, as we showed in Fig. 8. We also note that in traditional audio, a PESQ score of 2 is considered fair to good in the presence of white noise; however our interference is real audio from ambient FM signals. What we hear is a composite signal, in which a listener can easily hear the backscattered audio at a PESQ value of two. We attach samples of overlay backscatter audio signals with PESQ values of 2.5, 2, 1.5 and 1 respectively in the following video for demonstration:
`https://smartcities.cs.washington.edu/pesq.mp4`

Audio with stereo backscatter. Next we show that we can backscatter audio information in the stereo stream to minimize the effect of the background FM audio. We run experiments on two different FM signals: a mono only FM station and a stereo news and information station. In the first case, in addition to the audio information, our backscatter device inserts the 19 kHz pilot signal to convert the ambient mono transmission into a stereo

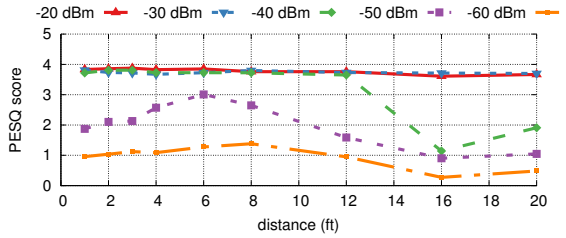


Figure 12: **PESQ w/ cooperative backscatter.** PESQ scores of speech received with cooperative cancellation techniques *w.r.t.* different received powers and distances.

FM signal. We use the same USRP transmitter setup to broadcast an audio signal recorded from a local mono FM station. In both these scenarios, we run the algorithm in §3.3 to cancel the underlying ambient FM audio. We repeat the experiments above and plot the results in Fig. 13a and Fig. 13b. The plots show the following:

- At high FM powers, the PESQ of stereo backscatter is much higher than overlay backscatter. This is because news FM stations underutilize the stereo stream and so the backscattered audio can be decoded with lower noise. At lower power numbers however, FM receivers cannot decode the pilot signal and default back to mono mode. As a result stereo backscatter requires high ambient FM power levels to operate.
- Fig. 13b shows the feasibility of transforming mono FM transmissions into stereo signals. Further, stereo backscatter of a mono FM station results in higher PESQ and can operate at a lower power of -40 dBm. This is because, unlike news stations that have some signal in the stereo stream, mono transmissions have no signals and therefore even less interference than the previous case.

Audio with cooperative backscatter. Finally, we evaluate cooperative backscatter using audio transmissions. Specifically, we use two Moto G1 smartphones to achieve a MIMO system and cancel the underlying ambient audio signals. The FM transmitter broadcasts at a frequency of 91.5 MHz and the backscatter device again shifts the transmission by 600 kHz to 92.1 MHz. We set the first smartphone to 92.1 MHz, and the second smartphone to 91.5 MHz and keep them equidistant from our backscatter prototype. We use the cancellation algorithm in §3.3 to decode the backscattered audio signals. We repeat the above set of experiments and plot the results in Fig. 12. The plots show that cooperative backscatter has high PESQ values of around 4 for different power values between -20 and -50 dBm. This is expected because the MIMO algorithm in §3.3 cancels the underlying audio signal. We also note that cooperative backscatter can operate with much weaker ambient FM signals (-50 dBm) than stereo backscatter (-40 dBm). This is because as explained earlier, radios do not operate in the stereo mode when the incoming FM signal is weak.

5.4 Using FM Receivers in Cars

We evaluate our backscatter system with an FM radio receiver built into a car in order to further demonstrate the potential for these techniques to enable connected city applications. The FM receivers built into cars have two distinct differences compared to smartphones. First, car antennas can be better optimized compared to phones as they have less space constraints, the body of the car can provide a large ground plane, and the placement and orientation of the antenna can be precisely defined and fixed unlike the loose wires used for headphone antennas. Because of this, we expect the RF performance of the car's antenna and radio receiver to be significantly better than the average smartphone. Second, although recent car models have begun to offer software such as Android Auto or Apple CarPlay, the vast majority of car stereos are still not programmable and therefore limited to using our overlay backscatter technique.

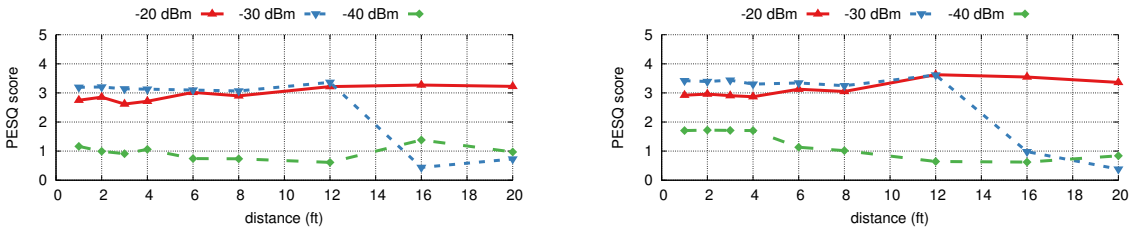
To test the performance of the car receiver with our backscatter system, we use a similar experimental setup to the one described above for the smartphone. Specifically, we place the backscatter antenna 12 ft away from the transmitting antenna which we configure to output a measured power, and evaluate the quality of the audio signal received in a 2010 Honda CRV versus range. Because the radio built into the car does not provide a direct audio output, we use a microphone to record the sound played by the car's speakers. To simulate a realistic use case we perform all experiments with the car's engines running and the windows closed. We backscatter the same signals used above to measure SNR and PESQ. Figure 14 shows the audio quality versus range for two different power values, demonstrating our system works well up to 60 ft.

6 Proof-of-concept Applications

We evaluate two proof-of-concept applications to show how FM backscatter applies to real world scenarios.

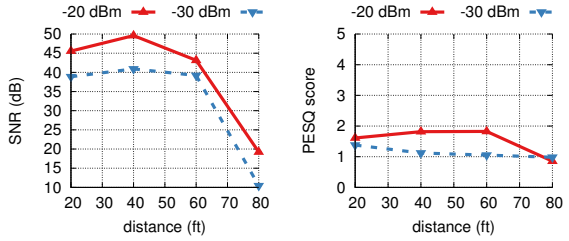
6.1 Talking Posters

Posters and billboards have long been a staple of advertising in public spaces. However these traditional methods of advertising rely entirely on a single graphic to attract attention and convey information. A poster capable of broadcasting to a smartphone could send audio, notifications and links for discounted event tickets. This added level of interactivity allows the advertisement to persist beyond a quick glance and allows the advertiser to take advantage of smartphone functionalities, for example to give directions to an event.



(a) Stereo station (b) Mono station

Figure 13: **PESQ w/ stereo backscatter.** PESQ scores of speech received with stereo backscatter techniques *w.r.t.* different receiving powers and distances: (a) is for sending audio in the stereo stream of a stereo broadcast and (b) is for transforming a mono station into a stereo transmission.



(a) SNR (b) PESQ scores

Figure 14: **Overlay backscatter performance in a car.** SNR and PESQ scores recorded in a car /wrt different receiving power and distances.

We leverage the low power backscatter techniques described in this paper to show that posters can broadcast information directly to cars and smartphones in outdoor environments. To evaluate this, we design two poster form factor antennas. We fabricate a half wavelength dipole antenna on a bus stop size poster (40"x60"), as well as a bowtie antenna in the common Super A1 poster size (24"x36"). Both are fabricated by placing copper tape (Tapes Master) in the desired pattern onto a sheet of 45 lb poster paper. Fig. 15a shows the 24"x36" prototype.

To evaluate this system, we place our prototype poster antenna on the side of a real bus stop, as shown in Fig. 15b. We backscatter ambient radio signals from a local public radio station broadcasting news and information at a frequency of 94.9 MHz. At the poster location we measure an ambient signal power of -35 dBm to -40 dBm. We test both data and audio transmissions using overlay backscatter. We create our backscatter signal at 95.3 MHz. A user standing next to the poster wearing a pair of headphones connected to a Moto G2 smartphone records these backscattered signals. Our results show that we can decode data at 100 bps at distances of up to ten feet. Fig. 16 shows an advertisement for the band *Simply Three* as a notification on a smartphone. We also overlay a snippet from the band's music on top of the ambient news signals and record the audio decoded by the phone at a distance of 4 ft from the poster.

In addition to evaluating the poster with smartphones, we also evaluate this system using the FM radio receiver built into a car to further demonstrate how these poster and sign form factor devices could be used for connected city applications. We mount the poster on a wall 5 ft above the ground and park a 2010 Honda CRV 10 ft from the poster. To simulate a realistic use case in an urban environment, we position the poster on the side of the building without direct line of sight to the FM transmitter. We again backscatter the ambient radio signal available at 94.9 MHz to 95.3 MHz and use a microphone to record the audio played from the car's speakers. We attach sample clips of the above recordings at the following link: `https://smartcities.cs.washington.edu/apps.mp4`

6.2 Smart Fabric

There has been recent interest in smart fabrics that integrate computing, sensing and interaction capabilities into garments. Given their unique advantage of being in contact with the body, textiles are well suited for sensing vital signs including heart and breathing rate. In fact, clothing manufacturers have begun to explore integrating sensors into different types of clothing[22, 3]. More recently, Project Jacquard from Google [44] designs touch-sensitive textiles to enable novel interaction capabilities. We explore the potential of using FM radio backscatter to enable connectivity for smart fabrics. Low power connectivity can enable smart fabrics that do not require large batteries, which is useful for reducing form factor and improving durability.

We design a prototype of our backscatter system on a shirt as shown in Fig. 17a. Specifically, we use Ansys HFSS to design and simulate a meander dipole antenna small enough to fit on the front of a standard 15 in wide adult small T-shirt. Building on the antenna designs in [33, 26], we fabricate our prototype on a 100% cotton t-shirt by machine sewing patterns of a conductive 3 ply thread made out of 316L stainless steel fibers [17] which does not oxidize even after repeated use or washing.

Wearable systems suffer from losses such poor antenna performance in close proximity to the human body and must perform consistently while a user is in motion. To evaluate whether our backscatter system is suitable for

(a) (b)

Figure 15: **Talking Poster application.** The figures show both a close up of our poster form factor antenna as well as its deployment at a bus stop.

these applications, we connect our backscatter switch to the prototype shirt antenna and use it to transmit data at both 100 bps and 1.6 kbps. We perform this experiment in an outdoor environment in which the prototype antenna receives ambient radio signals at a level of -35 dBm to -40 dbm. Fig. 17b compares the bit error rate when the user is standing still, running (2.2 m/s) or walking (1 m/s). The plot shows that at a bit rate of 1.6 kbps while using MRC, the BER was roughly 0.02 while standing and increases with motion. However at a lower bit rate of 100 bps, the BER was less than 0.005 even when the user was running. This demonstrates that FM radio backscatter can be used for smart fabric applications.

7 Related Work

Our work is related to RFID systems [32, 10, 48, 51, 34, 41] that use expensive readers as dedicated signal sources. The challenge in using RFID systems outdoors is the cost of deploying and maintaining the RFID reader infrastructure. This has in general tempered the adoption of RFID systems in applications beyond inventory tracking.

Ambient backscatter [40, 42] enables two RF-powered devices to communicate with each other by scattering ambient TV signals. In contrast, our focus is to transmit to mobile devices such as smartphones using backscatter communication. Since smartphones do not have TV receivers, TV signals are unsuitable for our purposes. Furthermore, given the transition to digital TV, the number of available broadcast TV channels has been declining over the years [27]. So it is unlikely that future smartphones will incorporate TV receivers.

[37] backscatters transmissions from a Wi-Fi router, which can be decoded on existing Wi-Fi chipsets using changes to the per-packet CSI/RSSI values. BackFi [43] improved the rate of this communication with a full-duplex radio at the router to cancel the high-power Wi-Fi transmissions from the reader and decode the weak backscattered signal. Passive Wi-Fi [38] and [29] demonstrated the feasibility of generating 802.11b and Bluetooth transmissions using backscatter communication. Both these systems require infrastructure support in terms of a plugged in device that transmits a single tone signal. Given the scarcity of Wi-Fi outdoors as well as the challenges in deploying plugged-in devices, these approaches are not applicable in outdoor environments.

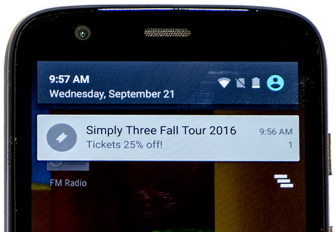


Figure 16: Example of a notification sent by a poster advertising discounted tickets to a local concert.

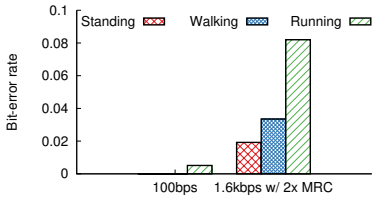
More recently, work on FS-backscatter [49] and Interscatter [36] demonstrate that one can backscatter either Bluetooth or Wi-Fi transmissions from a device (e.g., smartphone or router) and decode the backscattered signals on another device (e.g., smart watch). These approaches require two different devices to be carried by the user, which is not convenient. Further, given the lukewarm adoption of smart watches [21], it is important to explore the solution space for techniques that can work without them. In contrast, this paper shows that one can leverage the ambient FM signals in outdoor environments as a signal source to enable transmissions from everyday objects to mobile devices such as smartphones.

Bluetooth [4], NFC [50] and QR-codes [39] are alternate technologies to enable connectivity. Our backscatter design is orders of magnitude lower power and cheaper (5-10 cents [9]) than Bluetooth since it does not require generating an RF signal. Further, it does not require pointing the phone camera toward the QR-codes and has a better range than NFC, which is limited to a few cm. Additionally, the ranges we achieve allow multiple people to easily view and interact with the poster at once unlike NFC or QR codes.

Finally, FCC regulations allowing weak unlicensed transmitters in FM bands [30] led to the proliferation of personal FM transmitters [19, 20]. These transmitters however consume orders of magnitude more power and are therefore not suitable for our target applications.

8 Discussion and Conclusion

This paper opens a new direction for backscatter research by showing for the first time that ambient FM radio signals can be used as signal sources for backscatter communication. In this section, we discuss various aspects of this



(a) (b)

Figure 17: **Smart fabric application.** (a) shows a cotton shirt with an antenna sewn onto the front with conductive thread and (b) shows BER for the shirt in various mobility scenarios.

technology and outline avenues for future research.

Multiple backscatter devices. Since the range is currently limited to 4–60 feet, multiple devices that are spread out can concurrently operate without interference. For backscattering devices that are closer to each other, one can set f_{back} to different values so that the backscattered signals lie in different unused FM bands. We can also use MAC protocols similar to the Aloha protocol [25] to enable multiple devices to share the same FM band.

Improving data ranges and capabilities. We can use coding [42] to improve the FM backscatter range. We can also make the data transmission embedded in the underlying FM signals to be inaudible from a user perspective by using recent techniques for imperceptible data transmission in audible audio frequencies [47].

Power harvesting. We can explore powering these devices by harvesting from ambient RF signals such as FM or TV [42, 45] or using solar energy that is often plentiful in outdoor environments. We note that the power requirements could further be reduced by duty cycling transmissions. For example, a poster with an integrated motion sensor could transmit only when a person approaches.

Potpourri. One can also explore dual antenna designs that use Wi-Fi as an RF source indoors and switch to FM signals in outdoor environments. To enable the link from the phone to the backscatter device, we can use the ultra low-power Wi-Fi ON-OFF keying approach described in [36]. Finally, many phones use wired headphones as FM radio antennas. Although Apple recently removed the standard 3.5 mm headphone connector from its latest iPhone model, Apple offers an adapter to connect wired headphones [2]. Thus, future adaptors could be designed for use with headphones as FM antennas.

9 Acknowledgments.

We thank Deepak Ganesan for his helpful feedback on the paper. This work was funded in part by awards from the National Science Foundation (CNS–1452494, CNS–1407583, CNS–1305072) and Google Faculty Research Awards.

References

[1] AN192 A Complete FM Radio on a Chip. http://www.tel.uva.es/personales/tri/radio_TDA7000.pdf.

[2] Apple Lightning to 3.5 mm Headphone Jack Adapter. http://www.apple.com/shop/product/MMX62AM/A/lightning-to-35-mm-headphone-jack-adapter.

[3] Athos Men's Upper Body Kit. https://www.liveathos.com/products/mens-upper-body-kit.

[4] Beacons: What they are, how they work, and why apple's ibeacon technology is ahead of the pack. http://www.businessinsider.com/beacons-and-ibeacons-create-a-new-market-2013-12.

[5] Cadence rfspectre. http://www.cadence.com/products/rf/spectre_rf_simulation/pages/default.aspx.

[6] Efl700a39. http://www.st.com/content/ccc/resource/technical/document/datasheet/cd/ac/89/0b/b4/8e/43/0b/CD00270103.pdf/files/CD00270103.pdf/jcr:content/translations/en.CD00270103.pdf.

[7] FM Fool. www.fmfool.com/.

[8] Free radio on my phone. http://freeradioonmyphone.org/.

[9] How much does an rfid tag cost today? https://www.rfidjournal.com/faq/show?85.

[10] Impinj: Delivering item intelligence for the internet of things. http://www.impinj.com/.

[11] Motorola FM Radio. https://play.google.com/store/apps/details?id=com.motorola.fmplayer&hl=en.

[12] Panasonic cr2032. https://na.industrial.panasonic.com/sites/default/pidsa/files/crseries_datasheets_merged.pdf.

[13] Radio Locator. http://radio-locator.com/.

[14] Si4702/03-C19 Data Sheet. https://www.sparkfun.com/datasheets/BreakoutBoards/Si4702-03-C19-1.pdf.

[15] Si4712/13-b30 datasheet. http://www.silabs.com/documents/public/data-sheets/Si4712-13-B30.pdf.

[16] Silicon labs si4713-b30-gmr. http://www.digikey.com/product-detail/en/silicon-labs/SI4713-B30-GMR/SI4713-B30-GMR-ND/2278325.

[17] Stainless Medium Conductive Thread - 3 ply. https://play.google.com/store/apps/details?id=com.motorola.fmplayer&hl=en.

[18] Ti cc2541 datasheet. http://www.ti.com/lit/ds/symlink/cc2541.pdf.

[19] URed FM Transmitter. http://www.uredbrand.com/.

[20] VicTsing FM Transmitter. http://www.victsing.com/category/fm-transmitter/.

[21] Wearables are making gains, but they still face major barriers to adoption. http://www.businessinsider.com/wearables-adoption-barriers-what-keeps-people-from-buying-smartwatch-fitness-bands-2015-6.

[22] The best smart clothing: From biometric shirts to contactless payment jackets, 2016. http://www.wareable.com/smart-clothing/best-smart-clothing.

[23] Verizon galaxy s7 and s7 edge updates introduce fm radio, hd voice improvements, 2016. http://www.droid-life.com/2016/09/01/verizon-galaxy-s7-s7-edge-updates-introduce-fm-radio/.

[24] Your phone has an fm chip. so why canâĂŹt you listen to the radio?, 2016. http://www.wired.com/2016/07/phones-fm-chips-radio-smartphone/.

[25] N. Abramson. The aloha system: another alternative for computer communications. In *Proceedings of the November 17-19, 1970, fall joint computer conference*, pages 281–285. ACM, 1970.

[26] M. Ali and I. Khan. Wearable antenna design for fm radio. *Arabian Journal for Science and Engineering*, 39(8):6189–6195, 2014.

[27] P. Bahl, R. Chandra, T. Moscibroda, R. Murty, and M. Welsh. White space networking with wi-fi like connectivity. *ACM SIGCOMM Computer Communication Review*, 39(4):27–38, 2009.

[28] J. R. Carson. Notes on the theory of modulation. *Proceedings of the Institute of Radio Engineers*, 10(1):57–64, 1922.

[29] J. Ensworth and M. Reynolds. Every smart phone is a backscatter reader: Modulated backscatter compatibility with bluetooth 4.0 low energy (ble) devices. In *RFID, 2015 IEEE International Conference on*.

[30] FCC. Permitted Forms of Low Power Broadcast Operation. *"Public Notice 14089"*.

[31] FCC. Subpart B - FM Broadcast Stations. *"Code of Federal Regulations, Title 47 - Telecommunication, Chapter 1 - FCC, Subchapter C - Broadcast Radio Services, Part 73 - Radio Broadcast Services"*.

[32] J. Gummeson, P. Zhang, and D. Ganesan. Flit: A bulk transmission protocol for rfid-scale sensors. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 71–84, New York, NY, USA, 2012. ACM.

[33] O. Howlader. Miniaturized dipole antenna development for low frequency ground penetrating radar (gpr) system. *IET Conference Proceedings*, pages 1 (13 .)–1 (13 .)(1), January 2016.

[34] P. Hu, P. Zhang, and D. Ganesan. Laissez-faire: Fully asymmetric backscatter communication. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 255–267, New York, NY, USA, 2015. ACM.

[35] Y. Hu and P. C. Loizou. Evaluation of objective quality measures for speech enhancement. *IEEE Transactions on audio, speech, and language processing*, 16(1):229–238, 2008.

[36] V. Iyer, V. Talla, B. Kellogg, S. Gollakota, and J. Smith. Inter-technology backscatter: Towards internet connectivity for implanted devices. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 356–369, New York, NY, USA, 2016. ACM.

[37] B. Kellogg, A. Parks, S. Gollakota, J. R. Smith, and D. Wetherall. Wi-fi backscatter: Internet connectivity for rf-powered devices. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, 2014.

[38] B. Kellogg, V. Talla, S. Gollakota, and J. Smith. Passive wi-fi: Bringing low power to wi-fi transmissions. In *Usenix NSDI*, 2016.

[39] A. Lerner, A. Saxena, K. Ouimet, B. Turley, A. Vance, T. Kohno, and F. Roesner. Analyzing the use of quick response codes in the wild. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 359–374, New York, NY, USA, 2015. ACM.

[40] V. Liu, A. Parks, V. Talla, S. Gollakota, D. Wetherall, and J. R. Smith. Ambient backscatter: Wireless communication out of thin air. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 2013.

[41] J. Ou, M. Li, and Y. Zheng. Come and be served: Parallel decoding for cots rfid tags. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, pages 500–511, New York, NY, USA, 2015. ACM.

[42] A. N. Parks, A. Liu, S. Gollakota, and J. R. Smith. Turbocharging ambient backscatter communication. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, 2014.

[43] D. Pharadia, K. R. Joshi, M. Kotaru, and S. Katti. Backfi: High throughput wifi backscatter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015.

[44] I. Poupyrev, N.-W. Gong, S. Fukuhara, M. E. Karagozler, C. Schwesig, and K. E. Robinson. Project jacquard: Interactive digital textiles at scale. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 4216–4227, New York, NY, USA, 2016. ACM.

[45] V. Talla, B. Kellogg, B. Ransford, S. Naderiparizi, S. Gollakota, and J. R. Smith. Powering the next billion devices with wi-fi. In *ACM CoNEXT*, Dec. 2015.

[46] D. Tse and P. Viswanath. *Fundamentals of Wireless Communication*. Cambridge University Press, New York, NY, USA, 2005.

[47] Q. Wang, K. Ren, M. Zhou, T. Lei, D. Koutsonikolas, and L. Su. Messages behind the sound: Real-time hidden acoustic signal capture with smartphones. In *MOBICOM'16*.

[48] P. Zhang and D. Ganesan. Enabling bit-by-bit backscatter communication in severe energy harvesting environments. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 345–357, Berkeley, CA, USA, 2014. USENIX Association.

[49] P. Zhang, M. Rostami, P. Hu, and D. Ganesan. Enabling practical backscatter communication for on-body sensors. In *Proceedings of the ACM SIGCOMM 2016 Conference on SIGCOMM*, 2016.

[50] Y. Zhao, J. R. Smith, and A. Sample. Nfc-wisp: A sensing and computationally enhanced near-field rfid platform. In *2015 IEEE International Conference on RFID (RFID)*, pages 174–181, April 2015.

[51] Y. Zheng and M. Li. Read bulk data from computational rfids. *IEEE/ACM Transactions on Networking*, PP(99):1–1, 2016.

Prio: Private, Robust, and Scalable Computation of Aggregate Statistics

Henry Corrigan-Gibbs and Dan Boneh
Stanford University

Abstract. This paper presents Prio, a privacy-preserving system for the collection of aggregate statistics. Each Prio client holds a private data value (e.g., its current location), and a small set of servers compute statistical functions over the values of all clients (e.g., the most popular location). As long as at least one server is honest, the Prio servers learn nearly nothing about the clients' private data, except what they can infer from the aggregate statistics that the system computes. To protect functionality in the face of faulty or malicious clients, Prio uses *secret-shared non-interactive proofs* (SNIPs), a new cryptographic technique that yields a hundred-fold performance improvement over conventional zero-knowledge approaches. Prio extends classic private aggregation techniques to enable the collection of a large class of useful statistics. For example, Prio can perform a least-squares regression on high-dimensional client-provided data without ever seeing the data in the clear.

1 Introduction

Our smartphones, cars, and wearable electronics are constantly sending telemetry data and other sensor readings back to cloud services. With these data in hand, a cloud service can compute useful *aggregate statistics* over the entire population of devices. For example, navigation app providers collect real-time location data from their users to identify areas of traffic congestion in a city and route drivers along the least-crowded roads [80]. Fitness tracking services collect information on their users' physical activity so that each user can see how her fitness regimen compares to the average [75]. Web browser vendors collect lists of unusually popular homepages to detect homepage-hijacking adware [57].

Even when a cloud service is only interested in learning aggregate statistics about its user population as a whole, such services often end up collecting private data from each client and storing it for aggregation later on. These centralized caches of private user data pose severe security and privacy risks: motivated attackers may steal and disclose clients' sensitive information [84, 117], cloud services may misuse the clients' information for profit [112], and intelligence agencies may appropriate the data for targeting or mass surveillance purposes [65].

To ameliorate these threats, major technology companies, including Apple [72] and Google [57, 58], have deployed privacy-preserving systems for the collection of user data. These systems use a "randomized response" mechanism to achieve differential privacy [54, 118]. For example, a mobile phone vendor may want to learn how many of its phones have a particular uncommon but sensitive app installed (e.g., the AIDSinfo app [113]). In the simplest variant of this approach, each phone sends the vendor a bit indicating whether it has the app installed, except that the phone flips its bit with a fixed probability $p < 0.5$. By summing a large number of these noisy bits, the vendor can get a good estimate of the true number of phones that are running the sensitive app.

This technique scales very well and is robust even if some of the phones are malicious—each phone can influence the final sum by ± 1 at most. However, randomized-response-based systems provide relatively *weak privacy* guarantees: every bit that each phone transmits leaks some private user information to the vendor. In particular, when $p = 0.1$ the vendor has a good chance of seeing the correct (unflipped) user response. Increasing the noise level p decreases this leakage, but adding more noise also decreases the accuracy of the vendor's final estimate. As an example, assume that the vendor collects randomized responses from one million phones using $p = 0.49$, and that 1% of phones have the sensitive app installed. Even with such a large number of responses, the vendor will incorrectly conclude that *no phones* have the app installed roughly one third of the time.

An alternative approach to the data-collection problem is to have the phones send *encryptions* of their bits to a set of servers. The servers can sum up the encrypted bits and decrypt only the final sum [48, 56, 81, 92, 99, 100]. As long as all servers do not collude, these encryption-based systems provide much *stronger privacy* guarantees: the system leaks nothing about a user's private bit to the vendor, except what the vendor can infer from the final sum. By carefully adding structured noise to the final sum, these systems can provide differential privacy as well [56, 92, 107].

However, in gaining this type of privacy, many secret-sharing-based systems sacrifice *robustness*: a malicious client can send the servers an encryption of a large integer value v instead of a zero/one bit. Since the client's value v is encrypted, the servers cannot tell from inspecting the ciphertext that $v > 1$. Using this approach, a single malicious client can increase the final sum by v, instead of by 1.

Clients often have an incentive to cheat in this way: an app developer could use this attack to boost the perceived popularity of her app, with the goal of getting it to appear on the app store's home page. It is possible to protect against these attacks using zero-knowledge proofs [107], but these protections destroy *scalability*: checking the proofs requires heavy public-key cryptographic operations at the servers and can increase the servers' workload by orders of magnitude.

In this paper, we introduce Prio, a system for private aggregation that resolves the tension between privacy, robustness, and scalability. Prio uses a small number of servers; as long as one of the Prio servers is honest, the system leaks nearly nothing about clients' private data (in a sense we precisely define), except what the aggregate statistic itself reveals. In this sense, Prio provides a strong form of cryptographic *privacy*. This property holds even against an adversary who can observe the entire network, control all but one of the servers, and control a large number of clients.

Prio also maintains *robustness* in the presence of an unbounded number of malicious clients, since the Prio servers can detect and reject syntactically incorrect client submissions in a privacy-preserving way. For instance, a car cannot report a speed of 100,000 km/h if the system parameters only allow speeds between 0 and 200 km/h. Of course, Prio cannot prevent a malicious client from submitting an untruthful data value: for example, a faulty car can always misreport its actual speed.

To provide robustness, Prio uses a new technique that we call *secret-shared non-interactive proofs* (SNIPs). When a client sends an encoding of its private data to the Prio servers, the client also sends to each server a "share" of a proof of correctness. Even if the client is malicious and the proof shares are malformed, the servers can use these shares to collaboratively check—without ever seeing the client's private data in the clear—that the client's encoded submission is syntactically valid. These proofs rely only upon fast, information-theoretic cryptography, and require the servers to exchange only a few hundred bytes of information to check each client's submission.

Prio provides privacy and robustness without sacrificing *scalability*. When deployed on a collection of five servers spread around the world and configured to compute private sums over vectors of private client data, Prio imposes a 5.7× slowdown over a naïve data-collection system that provides no privacy guarantees whatsoever. In contrast, a state-of-the-art comparison system that uses client-generated non-interactive zero-knowledge proofs of correctness (NIZKs) [22, 103] imposes a 267× slowdown at the servers. Prio improves client performance as well: it is 50-100× faster than NIZKs and we estimate that it is three orders of magnitude faster than methods based on succinct non-interactive arguments of knowledge (SNARKs) [16,62,97]. The system is fast in absolute terms as well: when configured up to privately collect the distribution of responses to a survey with 434 true/false questions, the client performs only 26 ms of computation, and our distributed cluster of Prio servers can process each client submission in under 2 ms on average.

Contributions. In this paper, we:

- introduce *secret-shared non-interactive proofs* (SNIPs), a new type of information-theoretic zero-knowledge proof, optimized for the client/server setting,
- present *affine-aggregatable encodings*, a framework that unifies many data-encoding techniques used in prior work on private aggregation, and
- demonstrate how to combine these encodings with SNIPs to provide robustness and privacy in a large-scale data-collection system.

With Prio, we demonstrate that data-collection systems can simultaneously achieve strong privacy, robustness to faulty clients, and performance at scale.

2 System goals

A Prio deployment consists of a small number of infrastructure servers and a very large number of clients. In each time epoch, every client i in the system holds a private value x_i. The goal of the system is to allow the servers to compute $f(x_1, \ldots, x_n)$, for some aggregation function f, in a way that leaks as little as possible about each client's private x_i values to the servers.

Threat model. The parties to a Prio deployment must establish pairwise authenticated and encrypted channels. Towards this end, we assume the existence of a public-key infrastructure and the basic cryptographic primitives (CCA-secure public-key encryption [43, 108, 109], digital signatures [71], etc.) that make secure channels possible. We make no synchrony assumptions about the network: the adversary may drop or reorder packets on the network at will, and the adversary may monitor all links in the network. Low-latency anonymity systems, such as Tor [51], provide no anonymity in this setting, and Prio does not rely on such systems to protect client privacy.

Security properties. Prio protects client *privacy* as long as at least one server is honest. Prio provides *robustness* (correctness) only if all servers are honest. We summarize our security definitions here, but please refer to Appendix A for details.

Anonymity. A data-collection scheme maintains client anonymity if the adversary cannot tell which honest client submitted which data value through the system, even if the adversary chooses the honest clients' data values, controls all other clients, and controls all but one server. Prio always protects client anonymity.

Privacy. Prio provides f-privacy, for an aggregation function f, if an adversary, who controls any number of clients and all but one server, learns nothing about the honest clients' values x_i, except what she can learn from the value $f(x_1, \ldots, x_n)$ itself. More precisely, given $f(x_1, \ldots, x_n)$, every adversary controlling a proper subset of the servers, along with any number of clients, can simulate its view of the protocol run.

For many of the aggregation functions f that Prio implements, Prio provides strict f-privacy. For some aggregation functions, which we highlight in Section 5, Prio provides $\hat{f}$-privacy, where $\hat{f}$ is a function that outputs slightly more information than f. More precisely, $\hat{f}(x_1, \ldots, x_n) = \langle f(x_1, \ldots, x_n),\ L(x_1, \ldots, x_n)\rangle$ for some modest leakage function L.

Prio does not natively provide differential privacy [54], since the system adds no noise to the aggregate statistics it computes. In Section 7, we discuss when differential privacy may be useful and how we can extend Prio to provide it.

Robustness. A private aggregation system is robust if a coalition of malicious clients can affect the output of the system only by misreporting their private data values; a coalition of malicious clients cannot otherwise corrupt the system's output. For example, if the function $f(x_1, \ldots, x_n)$ counts the number of times a certain string appears in the set $\{x_1, \ldots, x_n\}$, then a single malicious client should be able to affect the count by at most one.

Prio is robust only against adversarial clients—*not* against adversarial servers. Although providing robustness against malicious servers seems desirable at first glance, doing so would come at privacy and performance costs, which we discuss in Appendix B. Since there could be millions of clients in a Prio deployment, and only a handful of servers (in fixed locations with known administrators), it may also be possible to eject faulty servers using out-of-band means.

3 A simple scheme

Let us introduce Prio by first presenting a simplified version of it. In this simple version, each client holds a one-bit integer x_i and the servers want to compute the sum of the clients' private values $\sum_i x_i$. Even this very basic functionality has many real-world applications. For example, the developer of a health data mobile app could use this scheme to collect the number of app users who have a certain medical condition. In this application, the bit x_i would indicate whether the user has the condition, and the sum over the x_is gives the count of affected users.

The public parameters for the Prio deployment include a prime p. Throughout this paper, when we write "$c = a+b \in \mathbb{F}_p$," we mean "$c = a+b \pmod{p}$." The simplified Prio scheme for computing sums proceeds in three steps:

1. **Upload.** Each client i splits its private value x_i into s shares, one per server, using a secret-sharing scheme. In particular, the client picks random integers $[x_i]_1, \ldots, [x_i]_s \in \mathbb{F}_p$, subject to the constraint: $x_i = [x_i]_1 + \cdots + [x_i]_s \in \mathbb{F}_p$. The client then sends, over an encrypted and authenticated channel, one share of its submission to each server.

2. **Aggregate.** Each server j holds an accumulator value $A_j \in \mathbb{F}_p$, initialized to zero. Upon receiving a share from the ith client, the server adds the uploaded share into its accumulator: $A_j \leftarrow A_j + [x_i]_j \in \mathbb{F}_p$.

3. **Publish.** Once the servers have received a share from each client, they publish their accumulator values. Computing the sum of the accumulator values $\sum_j A_j \in \mathbb{F}_p$ yields the desired sum $\sum_i x_i$ of the clients' private values, as long as the modulus p is larger than the number of clients (i.e., the sum $\sum_i x_i$ does not "overflow" the modulus).

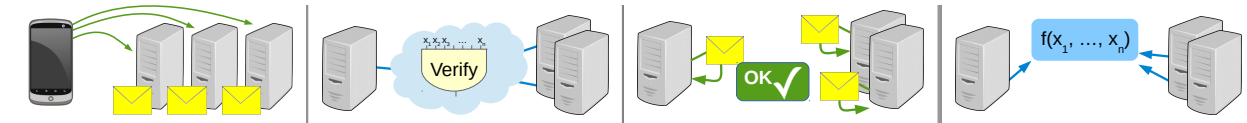
There are two observations we can make about this scheme. First, even this simple scheme provides privacy: the servers learn the sum $\sum_i x_i$ but they learn nothing else about the clients' private inputs. Second, the scheme does *not* provide robustness. A single malicious client can completely corrupt the protocol output by submitting (for example), a random integer $r \in \mathbb{F}_p$ to each server.

The core contributions of Prio are to improve this basic scheme in terms of security and functionality. In terms of security, Prio extends the simple scheme to provide robustness in the face of malicious clients. In terms of functionality, Prio extends the simple scheme to allow privacy-preserving computation of a wide array of aggregation functions (not just sum).

4 Protecting correctness with SNIPs

Upon receiving shares of a client's data value, the Prio servers need a way to check if the client-submitted value is well formed. For example, in the simplified protocol of Section 3, every client is supposed to send the servers the share of a value x such that $0 \le x \le 1$. However, since the client sends only a *single share* of its value x to each server—to preserve privacy—each server essentially receives an encrypted version of x and cannot unilaterally determine if x is well formed. In the more general setting, each Prio client submits to each server a share $[x]_i$ of a vector $x \in \mathbb{F}^L$, for some finite field $\mathbb{F}$. The servers hold a validation predicate $\mathsf{Valid}(\cdot)$, and should only accept the client's data submission if $\mathsf{Valid}(x) = 1$ (Figure 1).

To execute this check in Prio, we introduce a new cryptographic tool called *secret-shared non-interactive proofs* ("SNIPs"). With these proofs, the client can quickly prove



(a) The client sends a share of its encoded submission and SNIP proof to each server.

(b) The servers validate the client's SNIP proof to ensure that the submission is valid.

(c) If the checks pass, the servers update their local accumulators with the client-provided data.

(d) After accumulating many packets, the servers publish their accumulators to reveal the aggregate.

Figure 1: An overview of the Prio pipeline for processing client submissions.

to the servers that $\mathsf{Valid}(x) = 1$, for an arbitrary function Valid, without leaking anything else about x to the servers.

Building blocks. All arithmetic in this section takes place in a finite field $\mathbb{F}$, or modulo a prime p, if you prefer. We use a simple additive secret-sharing scheme over $\mathbb{F}$: to split a value $x \in \mathbb{F}$ into s shares, choose random values $([x]_1, \ldots, [x]_s) \in \mathbb{F}^s$ subject to the constraint that $x = \sum_i [x]_i \in \mathbb{F}$. In our notation, $[x]_i$ denotes the ith share of x. An adversary who gets hold of any subset of up to $s-1$ shares of x learns nothing, in an information-theoretic sense, about x from the shares.

This secret-sharing scheme is linear, which means that the servers can perform affine operations on shares without communicating. That is, by adding shares $[x]_i$ and $[y]_i$, the servers can locally construct shares $[x+y]_i$. Given a share $[x]_i$, the servers can also construct shares $[\alpha x+\beta]_i$, for any constants $\alpha, \beta \in \mathbb{F}$. (This is a classic observation from the multi-party computation literature [15].)

Our construction uses *arithmetic circuits*. An arithmetic circuit is like a boolean circuit except that it uses finite-field multiplication, addition, and multiplication-by-constant gates, instead of boolean AND, OR, and NOT gates. See Appendix C.1 for a formal definition.

4.1 Overview

A secret-shared non-interactive proof (SNIP) protocol consists of an interaction between a client (the prover) and multiple servers (the verifiers). At the start of the protocol:

- each server i holds a vector $[x]_i \in \mathbb{F}^L$,
- the client holds the vector $x = \sum_i [x]_i \in \mathbb{F}^L$, and
- all parties hold an arithmetic circuit representing a predicate $\mathsf{Valid} : \mathbb{F}^L \to \mathbb{F}$.

The client's goal is to convince the servers that $\mathsf{Valid}(x) = 1$, without leaking anything else about x to the servers. To do so, the client sends a proof string to each server. After receiving these proof strings, the servers gossip amongst themselves and then conclude either that $\mathsf{Valid}(x) = 1$ (the servers "accept x") or not (the servers "reject x").

For a SNIP to be useful in Prio, it must satisfy the following properties:

Correctness. If all parties are honest, the servers will accept x.

Soundness. If all servers are honest, and if $\mathsf{Valid}(x) \neq 1$, then for all malicious clients, even ones running in super-polynomial time, the servers will reject x with overwhelming probability. In other words, no matter how the client cheats, the servers will almost always reject x.

Zero knowledge. If the client and at least one server are honest, then the servers learn nothing about x, except that $\mathsf{Valid}(x) = 1$. More precisely, when $\mathsf{Valid}(x) = 1$, every proper subset of servers can simulate its view of the protocol execution.

These three security properties are nearly identical to the properties required of a zero-knowledge interactive proof system [70]. However, in the conventional zero-knowledge setting, there is a single prover and single verifier, whereas here we have a single prover (the client) and *many* verifiers (the servers).

Our contribution. We devise a SNIP that requires minimal server-to-server communication, is compatible with any public Valid circuit, and relies solely on fast, information-theoretic primitives. (We discuss how to hide the Valid circuit from the client in Section 4.4.)

To build the SNIP, we first generalize a "batch verification" technique of Ben-Sasson et al. [19] and then show how a set of servers can use it to verify an entire circuit computation by exchanging a only few field elements. We implement this last step with a new adaptation of Beaver's multi-party computation (MPC) protocol to the client/server setting [9].

Related techniques. Prior work has studied interactive proofs in both the many-prover [14, 60] and many-verifier settings [1, 10]. Prior many-verifier protocols require relatively expensive public-key primitives [1] or require an amount of server-to-server traffic that grows linearly in the size of the circuit for the Valid function [10]. In concurrent independent work, Boyle et al. [25] construct what we can view as a very efficient SNIP for a *specific* Valid function [25]. They also use a Beaver-style MPC multiplication; their techniques otherwise differ from ours.

4.2 Constructing SNIPs

To run the SNIP protocol, the client and servers execute the following steps:

Set-up. Let M be the number of multiplication gates in the arithmetic circuit for Valid. We work over a field $\mathbb{F}$ that is large enough to ensure that $2M \ll |\mathbb{F}|$.

Step 1: Client evaluation. The client evaluates the Valid circuit on its input x. The client thus knows the value that every wire in the circuit takes on during the computation of $\mathsf{Valid}(x)$. The client uses these wire values to construct three polynomials f, g, and h, which encode the values on the input and output wires of each of the M multiplication gates in the $\mathsf{Valid}(x)$ computation.

Specifically, if the input wire values to the t-th multiplication gate, in topological order from inputs to outputs, are u_t and v_t, then for all $1 \le t \le M$, we define f and g to be the lowest-degree polynomials such that $f(t) = u_t$ and $g(t) = v_t$. Then, we define the polynomial h as $h = f \cdot g$.

The polynomials f and g will have degree at most $M-1$, and the polynomial h will have degree at most $2M - 2$. Since $h(t) = f(t) \cdot g(t) = u_t \cdot v_t$ for all $t \in \{1, \ldots, M\}$, $h(t)$ is equal to the value of the output wire $(u_t \cdot v_t)$ of the t-th multiplication gate in the $\mathsf{Valid}(x)$ circuit.

In Step 1 of the checking protocol, the client executes the computation of $\mathsf{Valid}(x)$, uses polynomial interpolation to construct the polynomials f and g, and multiplies these polynomials to produce $h = f \cdot g$. The client then splits the coefficients of h using additive secret sharing and sends the ith share of the coefficients $[h]_i$ to server i.

Step 2: Consistency checking at the servers. Each server i holds a share $[x]_i$ of the client's private value x. Each server also holds a share $[h]_i$. Using $[x]_i$ and $[h]_i$, each server can—without communicating with the other servers—produce shares $[f]_i$ and $[g]_i$ of the polynomials f and g.

To see how, first observe that if a server has a share of every wire value in the circuit, it can construct $[f]_i$ and $[g]_i$ using polynomial interpolation. Next, realize that each server can reconstruct a share of every wire value in the circuit since each server:

- has a share of each of the input wire values ($[x]_i$),
- has a share of each wire value coming out of a multiplication gate (the value $[h]_i(t)$ is a share of the t-th such wire), and
- can derive all other wire value shares via affine operations on the wire value shares it already has.

Using these wire value shares, the servers use polynomial interpolation to construct $[f]_i$ and $[g]_i$.

If the client and servers have acted honestly up to this point, then the servers will now hold shares of polynomials f, g, and h such that $f \cdot g = h$.

In contrast, a malicious client could have sent the servers shares of a polynomial $\hat{h}$ such that, for some $t \in \{1, \ldots, M\}$, $\hat{h}(t)$ is *not* the value on the output wire in the t-th multiplication gate of the $\mathsf{Valid}(x)$ computation. In this case, the servers will reconstruct shares of polynomials $\hat{f}$ and $\hat{g}$ that might not be equal to f and g. We will then have with certainty that $\hat{h} \neq \hat{f} \cdot \hat{g}$. To see why, consider the least t_0 for which $\hat{h}(t_0) \neq h(t_0)$. For all $t \le t_0$, $\hat{f}(t) = f(t)$ and $\hat{g}(t) = g(t)$, by construction. Since

$$\hat{h}(t_0) \neq h(t_0) = f(t_0) \cdot g(t_0) = \hat{f}(t_0) \cdot \hat{g}(t_0),$$

it must be that $\hat{h}(t_0) \neq \hat{f}(t_0) \cdot \hat{g}(t_0)$, so $\hat{h} \neq \hat{f} \cdot \hat{g}$. (Ben-Sasson et al. [19] use polynomial identities to check the consistency of secret-shared values in a very different MPC protocol. Their construction inspired our approach.)

Step 3a: Polynomial identity test. At the start of this step, each server i holds shares $[\hat{f}]_i$, $[\hat{g}]_i$, and $[\hat{h}]_i$ of polynomials $\hat{f}$, $\hat{g}$, and $\hat{h}$. Furthermore, it holds that $\hat{f} \cdot \hat{g} = \hat{h}$ if and only if the servers collectively hold a set of wire value shares that, when summed up, equal the internal wire values of the $\mathsf{Valid}(x)$ circuit computation.

The servers now execute the Schwartz-Zippel randomized polynomial identity test [104, 126] to check whether this relation holds. The principle of the test is that if $\hat{f} \cdot \hat{g} \neq \hat{h}$, then the polynomial $\hat{f} \cdot \hat{g} - \hat{h}$ is a non-zero polynomial of degree at most $2M - 2$. Such a polynomial can have at most $2M - 2$ zeros in $\mathbb{F}$, so if we choose a random $r \in \mathbb{F}$ and evaluate $\hat{f}(r) \cdot \hat{g}(r) - \hat{h}(r)$, the servers will detect that $\hat{f} \cdot \hat{g} \neq \hat{h}$ with probability at least $1 - \frac{2M-2}{|\mathbb{F}|}$.

To execute the test, one of the servers samples a random value $r \in \mathbb{F}$. Each server i then evaluates her share of each of the three polynomials on the point r to get $[\hat{f}(r)]_i$, $[\hat{g}(r)]_i$, and $[\hat{h}(r)]_i$. The servers can perform this step locally, since polynomial evaluation requires only affine operations on shares.

Assume for a moment that each server i can multiply her shares $[\hat{f}(r)]_i$ and $[\hat{g}(r)]_i$ to produce a share $[\hat{f}(r) \cdot \hat{g}(r)]_i$. In this case, the servers can use a linear operation to get shares $\sigma_i = [\hat{f}(r) \cdot \hat{g}(r) - \hat{h}(r)]_i$. The servers then publish these σ_is and ensure that $\sum_i \sigma_i = 0 \in \mathbb{F}$, which implies that if $\hat{f}(r) \cdot \hat{g}(r) = \hat{h}(r)$. The servers reject the client's submission if $\sum_i \sigma_i \neq 0$.

Step 3b: Multiplication of shares. Finally, the servers must somehow multiply their shares $[\hat{f}(r)]_i$ and $[\hat{g}(r)]_i$ to get a share $[\hat{f}(r) \cdot \hat{g}(r)]_i$ without leaking anything to each other about the values $\hat{f}(r)$ and $\hat{g}(r)$. To do so, we adapt a multi-party computation (MPC) technique of Beaver [9]. The details of Beaver's MPC protocol are not critical here, but we include them for reference in Appendix C.2.

Beaver's result implies that if servers receive, from a trusted dealer, one-time-use shares $([a]_i, [b]_i, [c]_i) \in \mathbb{F}^3$ of random values such that $a \cdot b = c \in \mathbb{F}$ ("multiplication triples"), then the servers can very efficiently execute a

multi-party multiplication of a pair secret-shared values. Furthermore, the multiplication protocol is fast: it requires each server to broadcast a single message.

In the traditional MPC setting, the parties to the computation have to run an expensive cryptographic protocol to generate the multiplication triples themselves [46]. In our setting however, the client generates the multiplication triple on behalf of the servers: the client chooses $(a, b, c) \in \mathbb{F}^3$ such that $a \cdot b = c \in \mathbb{F}$, and sends shares of these values to each server. If the client produces shares of these values correctly, then the servers can perform a multi-party multiplication of shares to complete the correctness check of the prior section.

Crucially, *even if the client sends shares of an invalid multiplication triple to the servers*, the servers will still catch the cheating client with high probability. To see why: say that a cheating client sends the servers shares $([a]_i, [b]_i, [c]_i) \in \mathbb{F}^3$ such that $a \cdot b \neq c \in \mathbb{F}$. Then we can write $a \cdot b = (c + \alpha) \in \mathbb{F}$, for some constant $\alpha > 0$.

In this case, when the servers run Beaver's MPC multiplication protocol to execute the polynomial identity test, they will instead test whether $\hat{f}(r) \cdot \hat{g}(r) - \hat{h}(r) + \alpha = 0 \in \mathbb{F}$. (To confirm this, consult our summary of Beaver's protocol in Appendix C.2.) Since we only require soundness to hold if all servers are honest, we may assume that the client did not know the servers' random value r when the client generated its multiplication triple. This implies that r is distributed independently of α, and since we only require soundness to hold if the servers are honest, we may assume that r is sampled uniformly from $\mathbb{F}$ as well.

So, even if the client cheats, the servers will still be executing the polynomial identity test on a non-zero polynomial of degree at most $(2M - 2)$. The servers will thus catch a cheating client with probability at least $1 - \frac{2M-2}{|\mathbb{F}|}$.

Step 4: Output verification. If all servers are honest, at the start of the final step of the protocol, each server i will hold a set of shares of the values that the Valid circuit takes on during computation of $\mathsf{Valid}(x)$: $([w_1]_i, [w_2]_i, \dots)$. The servers already hold shares of the input wires of this circuit ($[x]_i$), so to confirm that $\mathsf{Valid}(x) = 1$, the servers need only publish their shares of the output wire. When they do, the servers can sum up these shares to confirm that the value on the output wire is equal to one, in which case it must be that $\mathsf{Valid}(x) = 1$, except with some small failure probability due to the polynomial identity test.

4.3 Security and efficiency

The correctness of the scheme follows by construction. To trick the servers into accepting a malformed submission, a cheating client must subvert the polynomial identity test. This bad event has probability at most $(2M-2)/|\mathbb{F}|$, where M is the number of multiplication gates in $\mathsf{Valid}(\cdot)$. By taking $|\mathbb{F}| \approx 2^{128}$, or repeating Step 3 a few times, we can make this failure probability extremely small.

		NIZK	SNARK	**Prio** (SNIP)
Client	Exps.	M	M	0
	Muls.	0	$M \log M$	$M \log M$
	Proof len.	M	1	M
Servers	Exps./Pairs.	M	1	0
	Muls.	0	M	$M \log M$
	Data transfer	M	1	1

Table 2: An asymptotic comparison of Prio with standard zero-knowledge techniques showing that Prio reduces the computational burden for clients and servers. The client holds a vector $x \in \mathbb{F}^M$, each server i holds a share $[x]_i$, and the client convinces the servers that each component of x is a 0/1 value in $\mathbb{F}$. We suppress the $\Theta(\cdot)$ notation for readability.

We require neither completeness nor soundness to hold in the presence of malicious *servers*, though we do require soundness against malicious clients. A malicious server can thus trick the honest servers into rejecting a well-formed client submission that they should have accepted. This is tantamount to the malicious server mounting a selective denial-of-service attack against the honest client. We discuss this attack in Section 7.

As long as there is at least one honest server, the properties of the secret sharing scheme guarantee that the dishonest servers gain no information—in an unconditional, information-theoretic sense—about the client's data values nor about the values on the internal wires in the $\mathsf{Valid}(x)$ circuit. Beaver's analysis [9] guarantees that the multiplication step leaks no information to the servers.

Efficiency. The remarkable property of this SNIP construction is that the server-to-server communication cost grows neither with the complexity of the verification circuit nor with the size of the value x (Table 2). The computation cost at the servers is essentially the same as the cost for each server to evaluate the Valid circuit locally.

That said, the client-to-server communication cost does grow linearly with the size of the Valid circuit. An interesting challenge would be to try to reduce the client's bandwidth usage without resorting to relatively expensive public-key cryptographic techniques [17, 18, 26, 41, 97].

4.4 Computation at the servers

Constructing the SNIP proof requires the client to compute $\mathsf{Valid}(x)$ on its own. If the verification circuit takes secret server-provided values as input, or is itself a secret belonging to the servers, then the client does not have enough information to compute $\mathsf{Valid}(x)$. For example, the servers could run a proprietary verification algorithm to detect spammy client submissions—the servers would want to run this algorithm without revealing it to the (possibly spam-producing) clients. To handle this use case,

the servers can execute the verification check themselves at a slightly higher cost. See Appendix D for details.

5 Gathering complex statistics

So far, we have developed the means to compute private sums over client-provided data (Section 3) and to check an arbitrary validation predicate against secret-shared data (Section 4). Combining these two ideas with careful data encodings, which we introduce now, allows Prio to compute more sophisticated statistics over private client data.

At a high level, each client first encodes its private data value in a prescribed way, and the servers then privately compute the sum of the encodings. Finally, the servers can decode the summed encodings to recover the statistic of interest. The participants perform this encoding and decoding via a mechanism we call affine-aggregatable encodings ("AFEs").

5.1 Affine-aggregatable encodings (AFEs)

In our setting, each client i holds a value $x_i \in \mathcal{D}$, where $\mathcal{D}$ is some set of data values. The servers hold an aggregation function $f : \mathcal{D}^n \to \mathcal{A}$, whose range is a set of aggregates $\mathcal{A}$. For example, the function f might compute the standard deviation of its n inputs. The servers' goal is to evaluate $f(x_1, \ldots, x_n)$ without learning the x_is.

An AFE gives an efficient way to encode the data values x_i such that it is possible to compute the value $f(x_1, \ldots, x_n)$ given only the *sum of the encodings* of $x_1, \ldots, x_n$. An AFE consists of three efficient algorithms $(\mathsf{Encode}, \mathsf{Valid}, \mathsf{Decode})$, defined with respect to a field $\mathbb{F}$ and two integers k and k', where $k' \leq k$:

- $\mathsf{Encode}(x)$: maps an input $x \in \mathcal{D}$ to its encoding in $\mathbb{F}^k$,
- $\mathsf{Valid}(y)$: returns true if and only if $y \in \mathbb{F}^k$ is a valid encoding of some data item in $\mathcal{D}$,
- $\mathsf{Decode}(\sigma)$: takes $\sigma = \sum_{i=1}^{n} \mathsf{Trunc}_{k'}(\mathsf{Encode}(x_i)) \in \mathbb{F}^{k'}$ as input, and outputs $f(x_1, \ldots, x_n)$. The $\mathsf{Trunc}_{k'}(\cdot)$ function outputs the first $k' \leq k$ components of its input.

The AFE uses all k components of the encoding in validation, but only uses k' components to decode σ. In many of our applications we have $k' = k$.

An AFE is *private with respect to a function* $\hat{f}$, or simply $\hat{f}$-private, if σ reveals nothing about $x_1, \ldots, x_n$ beyond what $\hat{f}(x_1, \ldots, x_n)$ itself reveals. More precisely, it is possible to efficiently simulate σ given only $\hat{f}(x_1, \ldots, x_n)$. Usually $\hat{f}$ reveals nothing more than the aggregation function f (i.e., the minimum leakage possible), but in some cases $\hat{f}$ reveals a little more than f.

For some functions f we can build more efficient f-private AFEs by allowing the encoding algorithm to be randomized. In these cases, we allow the decoding algorithm to return an answer that is only an approximation of f, and we also allow it to fail with some small probability.

Prior systems have made use of specific AFEs for sums [56, 86], standard deviations [100], counts [28, 92], and least-squares regression [82]. Our contribution is to unify these notions and to adopt existing AFEs to enable better composition with Prio's SNIPs. In particular, by using more complex encodings, we can reduce the size of the circuit for Valid, which results in shorter SNIP proofs.

AFEs in Prio: Putting it all together. The full Prio system computes $f(x_1, \ldots, x_n)$ privately as follows (see Figure 1): Each client encodes its data value x using the AFE Encode routine for the aggregation function f. Then, as in the simple scheme of Section 3, every client splits its encoding into s shares and sends one share to each of the s servers. The client uses a SNIP proof (Section 4) to convince the servers that its encoding satisfies the AFE Valid predicate.

Upon receiving a client's submission, the servers verify the SNIP to ensure that the encoding is well-formed. If the servers conclude that the encoding is valid, every server adds the first k' components of the encoding share to its local running accumulator. (Recall that k' is a parameter of the AFE scheme.) Finally, after collecting valid submissions from many clients, every server publishes its local accumulator, enabling anyone to run the AFE Decode routine to compute the final statistic in the clear. The formal description of the system is presented in Appendix G, where we also analyze its security.

Limitations. There exist aggregation functions for which all AFE constructions must have large encodings. For instance, say that each of n clients holds an integer x_i, where $1 \leq x_i \leq n$. We might like an AFE that computes the median of these integers $\{x_1, \ldots, x_n\}$, working over a field $\mathbb{F}$ with $|\mathbb{F}| \approx n^d$, for some constant $d \geq 1$.

We show that there is no such AFE whose encodings consist of $k' \in o(n/\log n)$ field elements. Suppose, towards a contradiction, that such an AFE did exist. Then we could describe any sum of encodings using at most $O(k' \log |\mathbb{F}|) = o(n)$ bits of information. From this AFE, we could build a single-pass, space-$o(n)$ streaming algorithm for computing the exact median of an n-item stream. But every single-pass streaming algorithm for computing the exact median over an n-item stream requires $\Omega(n)$ bits of space [74], which is a contradiction. Similar arguments may rule out space-efficient AFE constructions for other natural functions.

5.2 Aggregating basic data types

This section presents the basic affine-aggregatable encoding schemes that serve as building blocks for the more sophisticated schemes. In the following constructions, the clients hold data values $x_1, \ldots, x_n \in \mathcal{D}$, and our goal is to compute an aggregate $f(x_1, \ldots, x_n)$.

In constructing these encodings, we have two goals. The first is to ensure that the AFE leaks as little as possible about the x_is, apart from the value $f(x_1, \ldots, x_n)$ itself. The second is to minimize the number of multiplication gates in the arithmetic circuit for Valid, since the cost of the SNIPs grows with this quantity.

In what follows, we let λ be a security parameter, such as $\lambda = 80$ or $\lambda = 128$.

Integer sum and mean. We first construct an AFE for computing the sum of b-bit integers. Let $\mathbb{F}$ be a finite field of size at least $n2^b$. On input $0 \le x \le 2^b - 1$, the $\mathsf{Encode}(x)$ algorithm first computes the bit representation of x, denoted $(\beta_0, \beta_1, \ldots, \beta_{b-1}) \in \{0,1\}^b$. It then treats the binary digits as elements of $\mathbb{F}$, and outputs

$$\mathsf{Encode}(x) = (x, \beta_0, \ldots, \beta_{b-1}) \in \mathbb{F}^{b+1}.$$

To check that x represents a b-bit integer, the Valid algorithm ensures that each β_i is a bit, and that the bits represent x. Specifically, the algorithm checks that the following equalities hold over $\mathbb{F}$:

$$\mathsf{Valid}(\mathsf{Encode}(x)) = \left(x = \sum_{i=0}^{b-1} 2^i \beta_i\right) \wedge \bigwedge_{i=1}^{n} \Big[(\beta_i - 1) \cdot \beta_i = 0\Big].$$

The Decode algorithm takes the sum of encodings σ as input, truncated to only the first coordinate. That is, $\sigma = \sum_{i=1}^{n} \mathsf{Trunc}_1(\mathsf{Encode}(x_1)) = x_1 + \cdots + x_n$. This σ is the required aggregate output. Moreover, this AFE is clearly sum-private.

To compute the arithmetic mean, we divide the sum of integers by n over the rationals. Computing the product and geometric mean works in exactly the same matter, except that we encode x using b-bit logarithms.

Variance and STDDEV. Using known techniques [30, 100], the summation AFE above lets us compute the variance of a set of b-bit integers using the identity: $\mathrm{Var}(X) = \mathrm{E}[X^2] - (\mathrm{E}[X])^2$. Each client encodes its integer x as (x, x^2) and then applies the summation AFE to each of the two components. (The Valid algorithm also ensures that second integer is the square of the first.) The resulting two values let us compute the variance.

This AFE also reveals the expectation $\mathrm{E}[X]$. It is private with respect to the function $\hat{f}$ that outputs both the expectation and variance of the given set of integers.

Boolean OR and AND. When $\mathcal{D} = \{0,1\}$ and $f(x_1, \ldots, x_n) = \mathrm{OR}(x_1, \ldots, x_n)$ the encoding operation outputs an element of $\mathbb{F}_2^\lambda$ (i.e., a λ-bit bitstring) as:

$$\mathsf{Encode}(x) = \begin{cases} \lambda \text{ zeros} & \text{if } x = 0 \\ \text{a random element in } \mathbb{F}_2^\lambda & \text{if } x = 1. \end{cases}$$

The Valid algorithm outputs "1" always, since all λ-bit encodings are valid. The sum of encodings is simply the XOR of the n λ-bit encodings. The Decode algorithm takes as input a λ-bit string and outputs "0" if and only if its input is a λ-bit string of zeros. With probability $1 - 2^{-\lambda}$, over the randomness of the encoding algorithm, the decoding operation returns the boolean OR of the encoded values. This AFE is OR-private. A similar construction yields an AFE for boolean AND.

MIN and MAX. To compute the minimum and maximum of integers over a range $\{0, \ldots, B-1\}$, where B is small (e.g., car speeds in the range 0–250 km/h), the Encode algorithm can represent each integer in unary as a length-B vector of bits $(\beta_0, \ldots, \beta_{B-1})$, where $\beta_i = 1$ if and only if the client's value $x \le i$. We can use the bitwise-OR construction above to take the OR of the client-provided vectors—the largest value containing a "1" is the maximum. To compute the minimum instead, replace OR with AND. This is MIN-private, as in the OR protocol above.

When the domain is large (e.g., we want the MAX of 64-bit packet counters, in a networking application), we can get a c-approximation of the MIN and MAX using a similar idea: divide the range $\{0, \ldots, B-1\}$ into $b = \log_c B$ "bins" $[0, c), [c, c^2), \ldots, [c^{b-1}, B)$. Then, use the small-range MIN/MAX construction, over the b bins, to compute the approximate statistic. The output will be within a multiplicative factor of c of the true value. This construction is private with respect to the approximate MIN/MAX function.

Frequency count. Here, every client has a value x in a small set of data values $\mathcal{D} = \{0, \ldots, B-1\}$. The goal is to output a B-element vector v, where $v[i]$ is the number of clients that hold the value i, for every $0 \le i < B$.

Let $\mathbb{F}$ be a field of size at least n. The Encode algorithm encodes a value $x \in \mathcal{D}$ as a length-B vector $(\beta_0, \ldots, \beta_{B-1}) \in \mathbb{F}^B$ where $\beta_i = 1$ if $x = i$ and $\beta_i = 0$ otherwise. The Valid algorithm checks that each β value is in the set $\{0,1\}$ and that the sum of the βs is exactly one. The Decode algorithm does nothing: the final output is a length-B vector, whose ith component gives the number of clients who took on value i. Again, this AFE is private with respect to the function being computed.

The output of this AFE yields enough information to compute other useful functions (e.g., quantiles) of the distribution of the clients' x values. When the domain $\mathcal{D}$ is large, this AFE is very inefficient. In Appendix F, we give AFEs for approximate counts over large domains.

Sets. We can compute the intersection or union of sets over a small universe of elements using the boolean AFE operations: represent a set of B items as its characteristic vector of booleans, and compute an AND for intersection and an OR for union. When the universe is large, the approximate AFEs of Appendix F are more efficient.

		Workstation		Phone	
	Field size:	87-bit	265-bit	87-bit	265-bit
Mul. in field (μs)		1.013	1.485	11.218	14.930
Prio client time	$L = 10^1$	0.003	0.004	0.017	0.024
	$L = 10^2$	0.024	0.036	0.112	0.170
	$L = 10^3$	0.221	0.344	1.059	2.165

Table 3: Time in seconds for a client to generate a Prio submission of L four-bit integers to be summed at the servers. Averaged over eight runs.

5.3 Machine learning

We can use Prio for training machine learning models on private client data. To do so, we exploit the observation of Karr et al. [82] that a system for computing private sums can also privately train linear models. (In Appendix F, we also show how to use Prio to privately evaluate the R^2-coefficient of an existing model.) In Prio, we extend their work by showing how to perform these tasks while maintaining robustness against malicious clients.

Suppose that every client holds a data point (x, y) where x and y are b-bit integers. We would like to train a model that takes x as input and outputs a real-valued prediction $\hat{y}_i = M(x) \in \mathbb{R}$ of y. We might predict a person's blood pressure (y) from the number of steps they walk daily (x).

We wish to compute the least-squares linear fit $h(x) = c_0 + c_1 x$ over all of the client points. With n clients, the model coefficients c_0 and c_1 satisfy the linear relation:

$$\begin{pmatrix} n & \sum_{i=1}^n x_i \\ \sum_{i=1}^n x_i & \sum_{i=1}^n x_i^2 \end{pmatrix} \cdot \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^n y_i \\ \sum_{i=1}^n x_i y_i \end{pmatrix} \quad (1)$$

To compute this linear system in an AFE, every client encodes her private point (x, y) as a vector

$$(x, x^2, y, xy, \quad \beta_0, \ldots, \beta_{b-1}, \quad \gamma_0, \ldots, \gamma_{b-1}) \in \mathbb{F}^{2b+4},$$

where $(\beta_0, \ldots, \beta_{b-1})$ is the binary representation of x and $(\gamma_0, \ldots, \gamma_{b-1})$ is the binary representation of y. The validation algorithm checks that all the β and γ are in $\{0, 1\}$, and that all the arithmetic relations hold, analogously to the validation check for the integer summation AFE. Finally, the decoding algorithm takes as input the sum of the encoded vectors truncated to the first four components:

$$\sigma = (\ \textstyle\sum_{i=1}^n x, \quad \sum_{i=1}^n x^2, \quad \sum_{i=1}^n y, \quad \sum_{i=1}^n xy\ ),$$

from which the decoding algorithm computes the required real regression coefficients c_0 and c_1 using (1). This AFE is private with respect to the function that outputs the least-squares fit $h(x) = c_0 + c_1 x$, along with the mean and variance of the set $\{x_1, \ldots, x_n\}$.

When x and y are real numbers, we can embed the reals into a finite field $\mathbb{F}$ using a fixed-point representation, as long as we size the field large enough to avoid overflow.

The two-dimensional approach above generalizes directly to perform linear regression on d-dimensional feature vectors $\bar{x} = (x^{(1)}, \ldots, x^{(d)})$. The AFE yields a least-squares approximation of the form $h(\bar{x}) = c_0 + c_1 x^{(1)} + \cdots + c_d x^{(d)}$. The resulting AFE is private with respect to a function that reveals the least-square coefficients $(c_0, \ldots, c_d)$, along with the $d \times d$ covariance matrix $\sum_i \bar{x}_i \cdot (\bar{x}_i)^T$.

6 Evaluation

In this section, we demonstrate that Prio's theoretical contributions translate into practical performance gains. We have implemented a Prio prototype in 5,700 lines of Go and 620 lines of C (for FFT-based polynomial operations, built on the FLINT library [59]). Unless noted otherwise, our evaluations use an FFT-friendly 87-bit field. Our servers communicate with each other using Go's TLS implementation. Clients encrypt and sign their messages to servers using NaCl's "box" primitive, which obviates the need for client-to-server TLS connections. Our code is available online at https://crypto.stanford.edu/prio/.

We evaluate the SNIP-based variant of Prio (Section 4.1) and also the variant in which the servers keep the Valid predicate private ("Prio-MPC," Section 4.4). Our implementation includes three optimizations described in Appendix H. The first uses a pseudo-random generator (e.g., AES in counter mode) to reduce the client-to-server data transfer by a factor of roughly s in an s-server deployment. The second optimization allows the servers to verify SNIPs without needing to perform expensive polynomial interpolations. The third optimization gives an efficient way for the servers to compute the logical-AND of multiple arithmetic circuits to check that multiple Valid predicates hold simultaneously.

We compare Prio against a private aggregation scheme that uses non-interactive zero-knowledge proofs (NIZKs) to provide robustness. This protocol is similar to the "cryptographically verifiable" interactive protocol of Kursawe et al. [86] and has roughly the same cost, in terms of exponentiations per client request, as the "distributed decryption" variant of PrivEx [56]. We implement the NIZK scheme using a Go wrapper of OpenSSL's NIST P256 code [50]. We do not compare Prio against systems, such as ANONIZE [76] and PrivStats [100], that rely on an external anonymizing proxy to protect against a network adversary. (We discuss this related work in Section 8.)

6.1 Microbenchmarks

Table 3 presents the time required for a Prio client to encode a data submission on a workstation (2.4 GHz Intel Xeon E5620) and mobile phone (Samsung Galaxy SIII, 1.4 GHz Cortex A9). For a submission of 100 integers,

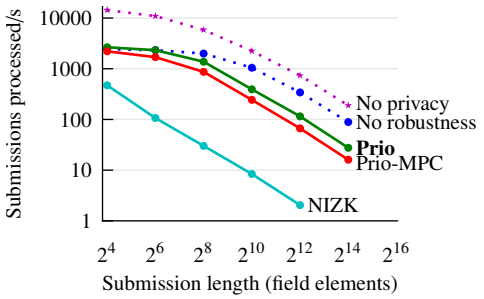


Figure 4: Prio provides the robustness guarantees of zero-knowledge proofs but at 20-50× less cost.

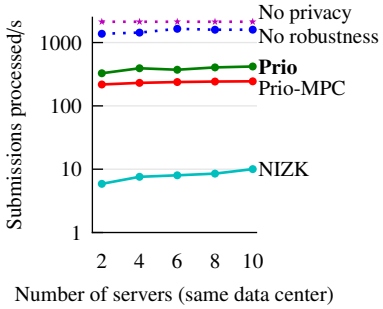


Figure 5: Prio is insensitive to the number of aggregation servers.

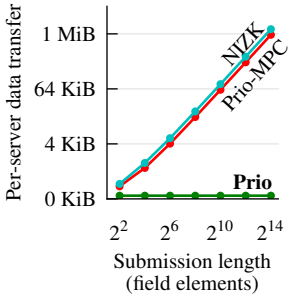


Figure 6: Prio's use of SNIPs (§4) reduces bandwidth consumption.

the client time is roughly 0.03 seconds on a workstation, and just over 0.1 seconds on a mobile phone.

To investigate the load that Prio places on the servers, we configured five Amazon EC2 servers (eight-core c3.2xlarge machines, Intel Xeon E5-2680 CPUs) in five Amazon data centers (N. Va., N. Ca., Oregon, Ireland, and Frankfurt) and had them run the Prio protocols. An additional three c3.2xlarge machines in the N. Va. data center simulated a large number of Prio clients. To maximize the load on the servers, we had each client send a stream of pre-generated Prio data packets to the servers over a single TCP connection. There is no need to use TLS on the client-to-server Prio connection because Prio packets are encrypted and authenticated at the application layer and can be replay-protected at the servers.

Figure 4 gives the throughput of this cluster in which each client submits a vector of zero/one integers and the servers sum these vectors. The "No privacy" line on the chart gives the throughput for a dummy scheme in which a single server accepts encrypted client data submissions directly from the clients with no privacy protection whatsoever. The "No robustness" line on the chart gives the throughput for a cluster of five servers that use a secret-sharing-based private aggregation scheme (*à la* Section 3) with no robustness protection. The five-server "No robustness" scheme is slower than the single-server "No privacy" scheme because of the cost of coordinating the processing of submissions amongst the five servers. The throughput of Prio is within a factor of 5× of the no-privacy scheme for many submission sizes, and Prio outperforms the NIZK-based scheme by more than an order of magnitude.

Finally, Figure 5 shows how the throughput of a Prio cluster changes as the number of servers increases, when the system is collecting the sum of 1,024 one-bit client-submitted integers, as in an anonymous survey application. For this experiment, we locate all of the servers in the same data center, so that the latency and bandwidth between each pair of servers is roughly constant. With more servers, an adversary has to compromise a larger number of machines to violate Prio's privacy guarantees.

Adding more servers barely affects the system's throughput. The reason is that we are able to load-balance the bulk of the work of checking client submissions across all of the servers. (This optimization is only possible because we require robustness to hold only if all servers are honest.) We assign a single Prio server to be the "leader" that coordinates the checking of each client data submission. In processing a single submission in an s-server cluster, the leader transmits s times more bits than a non-leader, but as the number of servers increases, each server is a leader for a smaller share of incoming submissions. The NIZK-based scheme also scales well: as the number of servers increases, the heavy computational load of checking the NIZKs is distributed over more machines.

Figure 6 shows the number of bytes each non-leader Prio server needs to transmit to check the validity of a single client submission for the two Prio variants, and for the NIZK scheme. The benefit of Prio is evident: the Prio servers transmit a constant number of bits per submission—*independent* of the size of the submission or complexity of the Valid routine. As the submitted vectors grow, Prio yields a 4,000-fold bandwidth saving over NIZKs, in terms of server data transfer.

6.2 Application scenarios

To demonstrate that Prio's data types are expressive enough to collect real-world aggregates, we have configured Prio for a few potential application domains.

Cell signal strength. A collection of Prio servers can collect the average mobile signal strength in each grid cell in a city without leaking the user's location history to the aggregator. We divide the geographic area into a km^2 grid—the number of grid cells depends on the city's size—and we encode the signal strength at the user's present location as a four-bit integer. (If each client only submits signal-strength data for a few grid cells in each protocol run, extra optimizations can reduce the client-to-server data transfer. See "Share compression" in Appendix F.)

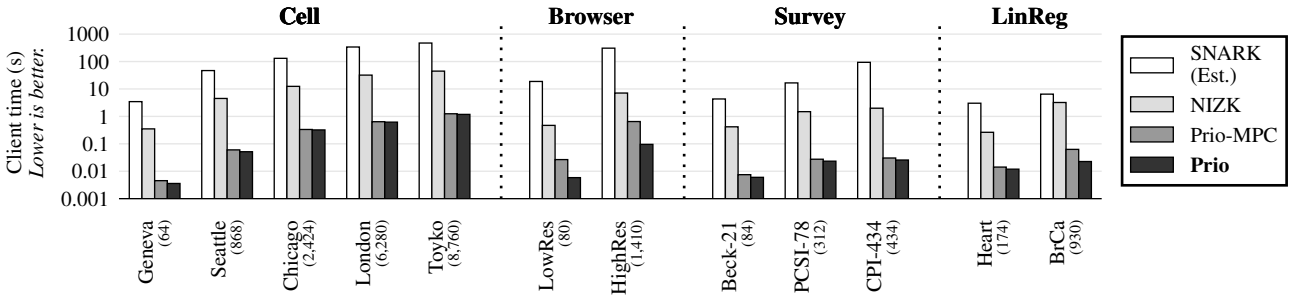


Figure 7: Client encoding time for different application domains when using Prio, a non-interactive zero-knowledge system (NIZK), or a SNARK-like system (estimated). Averaged over eight runs. The number of × gates in the Valid circuit is listed in parentheses.

Browser statistics. The Chromium browser uses the RAPPOR system to gather private information about its users [35,57]. We implement a Prio instance for gathering a subset of these statistics: average CPU and memory usage, along with the frequency counts of 16 URL roots. For collecting the approximate counts, we use the count-min sketch structure, described in Appendix F. We experiment with both low- and high-resolution parameters ($\delta = 2^{-10}$, $\epsilon = 1/10$; $\delta = 2^{-20}$, $\epsilon = 1/100$).

Health data modeling. We implement the AFE for training a regression model on private client data. We use the features from a preexisting heart disease data set (13 features of varying types: age, sex, cholesterol level, etc.) [78] and a breast cancer diagnosis data set (30 real-valued features using 14-bit fixed-point numbers) [120].

Anonymous surveys. We configure Prio to compute aggregates responses to sensitive surveys: we use the Beck Depression Inventory (21 questions on a 1-4 scale) [6], the Parent-Child Relationship Inventory (78 questions on a 1-4 scale) [63], and the California Psychological Inventory (434 boolean questions) [42].

Comparison to alternatives. In Figure 7, we compare the computational cost Prio places on the client to the costs of other schemes for protecting robustness against misbehaving clients, when we configure the system for the aforementioned applications. The fact that a Prio client need only perform a single public-key encryption means that it dramatically outperforms schemes based on public-key cryptography. If the Valid circuit has M multiplication gates, producing a discrete-log-based NIZK requires the client to perform $2M$ exponentiations (or elliptic-curve point multiplications). In contrast, Prio requires $O(M \log M)$ multiplications in a relatively small field, which is much cheaper for practical values of M.

In Figure 7, we give conservative estimates of the time required to generate a zkSNARK proof, based on timings of libsnark's [18] implementation of the Pinocchio system [97] at the 128-bit security level. These proofs have the benefit of being very short: 288 bytes, irrespective of the complexity of the circuit. To realize the benefit of these succinct proofs, the statement being proved must also be concise since the verifier's running time grows with the statement size. To achieve this conciseness in the Prio setting would require computing sL hashes "inside the SNARK," with s servers and submissions of length L.

We optimistically estimate that each hash computation requires only 300 multiplication gates, using a subset-sum hash function [2, 17, 67, 77], and we ignore the cost of computing the Valid circuit in the SNARK. We then use the timings from the libsnark paper to arrive at the cost estimates. Each SNARK multiplication gate requires the client to compute a number of exponentiations, so the cost to the client is large, though the proof is admirably short.

6.3 Machine learning

Finally, we perform an end-to-end evaluation of Prio when the system is configured to train a d-dimensional least-squares regression model on private client-submitted data, in which each training example consists of a vector of 14-bit integers. These integers are large enough to represent vital health information, for example.

In Figure 8, we show the client encoding cost for Prio, along with the no-privacy and no-robustness schemes described in Section 6.1. The cost of Prio's privacy and robustness guarantees amounts to roughly a 50× slowdown at the client over the no-privacy scheme due to the overhead of the SNIP proof generation. Even so, the absolute cost of Prio to the client is small—on the order of one tenth of a second.

Table 9 gives the rate at which the globally distributed five-server cluster described in Section 6.1 can process client submissions with and without privacy and robustness. The server-side cost of Prio is modest: only a 1-2× slowdown over the no-robustness scheme, and only a 5-15× slowdown over a scheme with *no privacy at all*. In contrast, the cost of robustness for the state-of-the-art NIZK schemes, per Figure 4, is closer to 100-200×.

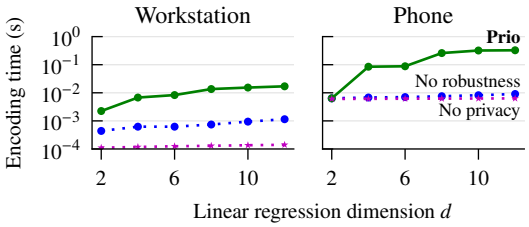


Figure 8: Time for a client to encode a submission consisting of d-dimensional training example of 14-bit values for computing a private least-squares regression.

	No privacy	No robustness		**Prio**		
d	Rate	Rate	Priv. cost	Rate	Robust. cost	Tot. cost
2	14,688	2,687	5.5×	2,541	1.1×	5.8×
4	15,426	2,569	6.0×	2,126	1.2×	7.3×
6	14,773	2,600	5.7×	1,897	1.4×	7.8×
8	15,975	2,564	6.2×	1,492	1.7×	10.7×
10	15,589	2,639	5.9×	1,281	2.1×	12.2×
12	15,189	2,547	6.0×	1,176	2.2×	12.9×

Table 9: The throughput, in client requests per second, of a global five-server cluster running a private d-dim. regression. We compare a scheme with no privacy, with privacy but no robustness, and Prio (with both).

7 Discussion

Deployment scenarios. Prio ensures client privacy as long as at least one server behaves honestly. We now discuss a number of deployment scenarios in which this assumption aligns with real-world incentives.

Tolerance to compromise. Prio lets an organization compute aggregate data about its clients without ever storing client data in a single vulnerable location. The organization could run all s Prio servers itself, which would ensures data privacy against an attacker who compromises up to $s - 1$ servers.

App store. A mobile application platform (e.g., Apple's App Store or Google's Play) can run one Prio server, and the developer of a mobile app can run the second Prio server. This allows the app developer to collect aggregate user data without having to bear the risks of holding these data in the clear.

Shared data. A group of s organizations could use Prio to compute an aggregate over the union of their customers' datasets, without learning each other's private client data.

Private compute services. A large enterprise can contract with an external auditor or a non-profit (e.g., the Electronic Frontier Foundation) to jointly compute aggregate statistics over sensitive customer data using Prio.

Jurisdictional diversity. A multinational organization can spread its Prio servers across different countries. If law enforcement agents seize the Prio servers in one country, they cannot deanonymize the organization's Prio users.

Common attacks. Two general attacks apply to *all systems*, like Prio, that produce exact (un-noised) outputs while protecting privacy against a network adversary. The first attack is a *selective denial-of-service attack*. In this attack, the network adversary prevents all honest clients except one from being able to contact the Prio servers [105]. In this case, the protocol output is $f(x_{\text{honest}}, x_{\text{evil}_1}, \ldots, x_{\text{evil}_n})$. Since the adversary knows the x_{evil} values, the adversary could infer part or all of the one honest client's private value x_{honest}.

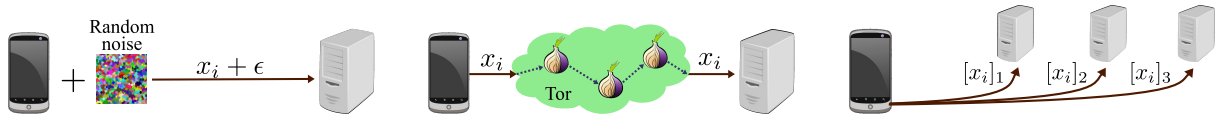
In Prio, we deploy the standard defense against this attack, which is to have the servers wait to publish the aggregate statistic $f(x_1, \ldots, x_n)$ until they are confident that the aggregate includes values from many honest clients. The best means to accomplish this will depend on the deployment setting.

One way is to have the servers keep a list of public keys of registered clients (e.g., the students enrolled at a university). Prio clients sign their submissions with the signing key corresponding to their registered public key and the servers wait to publish their accumulator values until a threshold number of registered clients have submitted valid messages. Standard defenses [3, 114, 124, 125] against Sybil attacks [52] would apply here.

The second attack is an *intersection attack* [20, 49, 83, 122]. In this attack, the adversary observes the output $f(x_1, \ldots, x_n)$ of a run of the Prio protocol with n honest clients. The adversary then forces the nth honest client offline and observes a subsequent protocol run, in which the servers compute $f(x'_1, \ldots, x'_{n-1})$. If the clients' values are constant over time ($x_i = x'_i$), then the adversary learns the difference $f(x_1, \ldots, x_n) - f(x_1, \ldots, x_{n-1})$, which could reveal client n's private value x_n (e.g., if f computes SUM).

One way for the servers to defend against the attack is to add differential privacy noise to the results before publishing them [54]. Using existing techniques, the servers can add this noise in a distributed fashion to ensure that as long as at least one server is honest, no server sees the un-noised aggregate [55]. The definition of differential privacy ensures that computed statistics are distributed approximately the same whether or not the aggregate includes a particular client's data. This same approach is also used in a system by Melis, Danezis, and De Cristofaro [92], which we discuss in Section 8.

Robustness against malicious servers. Prio only provides robustness when all servers are honest. Providing robustness in the face of faulty servers is obviously desirable, but we are not convinced that it is worth the security and performance costs. Briefly, providing robustness necessarily weakens the privacy guarantees that the system provides: if the system protects *robustness* in the presence of k faulty servers, then the system can protect *privacy*



(a) RAPPOR [57] provides differential privacy [54] (not information-theoretic privacy) by adding random noise to client submissions.

(b) ANONIZE [76] and PrivStats [100] rely on an anonymizing proxy, such as Tor [51], to protect privacy against network eavesdroppers.

(c) Prio and other schemes using secret sharing [30,48,56,79,86,92] offer ideal anonymity provided that the servers do not collude.

Figure 10: Comparison of techniques for anonymizing client data in private aggregation systems.

only against a coalition of at most $s - k - 1$ malicious servers. We discuss this issue further in Appendix B.

8 Related Work

Private data-collection systems [30, 48, 53, 56, 79, 86, 92] that use secret-sharing based methods to compute sums over private user data typically (a) provide no robustness guarantees in the face of malicious clients, (b) use expensive NIZKs to prevent client misbehavior, or (c) fail to defend privacy against actively malicious servers [33].

Other data-collection systems have clients send their private data to an aggregator through a general-purpose anonymizing network, such as a mix-net [27, 32, 47, 87] or a DC-net [31, 38–40, 110]. These anonymity systems provide strong privacy properties, but require expensive "verifiable mixing" techniques [8, 95], or require work at the servers that is *quadratic* in the number of client messages sent through the system [38, 121].

PrivStats [100] and ANONIZE [76] outsource to Tor [51] (or another low-latency anonymity system [61, 88, 101]) the work of protecting privacy against a network adversary (Figure 10). Prio protects against an adversary that can see and control the entire network, while Tor-based schemes succumb to traffic-analysis attacks [94].

In data-collection systems based on differential privacy [54], the client adds structured noise to its private value before sending it to an aggregating server. The added noise gives the client "plausible deniability:" if the client sends a value x to the servers, x could be the client's true private value, or it could be an unrelated value generated from the noise. Dwork et al. [55], Shi et al. [107], and Bassily and Smith [7] study this technique in a distributed setting, and the RAPPOR system [57, 58], deployed in Chromium, has put this idea into practice. A variant of the same principle is to have a trusted proxy (as in SuLQ [21] and PDDP [34]) or a set of minimally trusted servers [92] add noise to already-collected data.

The downside of these systems is that (a) if the client adds little noise, then the system does not provide much privacy, or (b) if the client adds a lot of noise, then low-frequency events may be lost in the noise [57]. Using server-added noise [92] ameliorates these problems.

In theory, secure multi-party computation (MPC) protocols [11, 15, 68, 90, 123] allow a set of servers, with some non-collusion assumptions, to privately compute *any* function over client-provided values. The generality of MPC comes with serious bandwidth and computational costs: evaluating the relatively simple AES circuit in an MPC requires the parties to perform many minutes or even hours of precomputation [44]. Computing a function f on millions of client inputs, as our five-server Prio deployment can do in tens of minutes, could potentially take an astronomical amount of time in a full MPC. That said, there have been great advances in practical general-purpose MPC protocols of late [12, 13, 23, 45, 46, 73, 89, 91, 93, 98]. General-purpose MPC may yet become practical for computing certain aggregation functions that Prio cannot (e.g., exact MAX), and some special-case MPC protocols [4, 29, 96] are practical today for certain applications.

9 Conclusion and future work

Prio allows a set of servers to compute aggregate statistics over client-provided data while maintaining client privacy, defending against client misbehavior, and performing nearly as well as data-collection platforms that exhibit neither of these security properties. The core idea behind Prio is reminiscent of techniques used in verifiable computation [16, 37, 62, 69, 97, 115, 116, 119], but in reverse—the client proves to a set of servers that it computed a function correctly. One question for future work is whether it is possible to efficiently extend Prio to support combining client encodings using a more general function than summation, and what more powerful aggregation functions this would enable. Another task is to investigate the possiblity of shorter SNIP proofs: ours grow linearly in the size of the Valid circuit, but sublinear-size information-theoretic SNIPs may be feasible.

Acknowledgements. We thank the anonymous NSDI reviewers for an extraordinarily constructive set of reviews. Jay Lorch, our shepherd, read two drafts of this paper and gave us pages and pages of insightful recommendations and thorough comments. It was Jay who suggested using Prio to privately train machine learning models, which became the topic of Section 5.3. Our colleagues, including David Mazières, David J. Wu, Dima Kogan, George Danezis, Phil Levis, Matei Zaharia, Saba Eskandarian, Sebastian Angel, and Todd Warszawski gave critical feedback that improved the content and presentation of the work. Any remaining errors in the paper are, of course, ours alone. This work received support from NSF, DARPA, the Simons Foundation, an NDSEG Fellowship, and ONR. Opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

[1] Abe, M., Cramer, R., and Fehr, S. Non-interactive distributed-verifier proofs and proving relations among commitments. In *ASIACRYPT* (2002), pp. 206–224.

[2] Ajtai, M. Generating hard instances of lattice problems. In *STOC* (1996), ACM, pp. 99–108.

[3] Alvisi, L., Clement, A., Epasto, A., Lattanzi, S., and Panconesi, A. SoK: The evolution of Sybil defense via social networks. In *Security and Privacy* (2013), IEEE, pp. 382–396.

[4] Applebaum, B., Ringberg, H., Freedman, M. J., Caesar, M., and Rexford, J. Collaborative, privacy-preserving data aggregation at scale. In *PETS* (2010), Springer, pp. 56–74.

[5] Archer, B., and Weisstein, E. W. Lagrange interpolating polynomial. http://mathworld.wolfram.com/LagrangeInterpolatingPolynomial.html. Accessed 16 September 2016.

[6] Association, A. P. Beck depression inventory. http://www.apa.org/pi/about/publications/caregivers/practice-settings/assessment/tools/beck-depression.aspx. Accessed 15 September 2016.

[7] Bassily, R., and Smith, A. Local, private, efficient protocols for succinct histograms. In *STOC* (2015), ACM, pp. 127–135.

[8] Bayer, S., and Groth, J. Efficient zero-knowledge argument for correctness of a shuffle. In *EUROCRYPT* (2012), Springer, pp. 263–280.

[9] Beaver, D. Efficient multiparty protocols using circuit randomization. In *CRYPTO* (1991), Springer, pp. 420–432.

[10] Beaver, D. Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority. *Journal of Cryptology 4*, 2 (1991), 75–122.

[11] Beaver, D., Micali, S., and Rogaway, P. The round complexity of secure protocols. In *STOC* (1990), ACM, pp. 503–513.

[12] Bellare, M., Hoang, V. T., Keelveedhi, S., and Rogaway, P. Efficient garbling from a fixed-key blockcipher. In *Security and Privacy* (2013), IEEE, pp. 478–492.

[13] Ben-David, A., Nisan, N., and Pinkas, B. FairplayMP: a system for secure multi-party computation. In *CCS* (2008), ACM, pp. 257–266.

[14] Ben-Or, M., Goldwasser, S., Kilian, J., and Wigderson, A. Multi-prover interactive proofs: How to remove intractability assumptions. In *STOC* (1988), ACM, pp. 113–131.

[15] Ben-Or, M., Goldwasser, S., and Wigderson, A. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC* (1988), ACM, pp. 1–10.

[16] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., and Virza, M. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *CRYPTO*. Springer, 2013, pp. 90–108.

[17] Ben-Sasson, E., Chiesa, A., Tromer, E., and Virza, M. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO* (2014), Springer, pp. 276–294.

[18] Ben-Sasson, E., Chiesa, A., Tromer, E., and Virza, M. Succinct non-interactive zero knowledge for a von Neumann architecture. In *USENIX Security* (2014), pp. 781–796.

[19] Ben-Sasson, E., Fehr, S., and Ostrovsky, R. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In *CRYPTO*. Springer, 2012, pp. 663–680.

[20] Berthold, O., and Langos, H. Dummy traffic against long term intersection attacks. In *Workshop on Privacy Enhancing Technologies* (2002), Springer, pp. 110–128.

[21] Blum, A., Dwork, C., McSherry, F., and Nissim, K. Practical privacy: the SuLQ framework. In *PODS* (2005), ACM, pp. 128–138.

[22] Blum, M., Feldman, P., and Micali, S. Non-interactive zero-knowledge and its applications. In *STOC* (1988), ACM, pp. 103–112.

[23] Bogetoft, P., Christensen, D. L., Damgard, I., Geisler, M., Jakobsen, T., Krøigaard, M., Nielsen, J. D., Nielsen, J. B., Nielsen, K., Pagter, J., Schwartzbach, M., and Toft, T. Multiparty computation goes live. In *Financial Cryptography* (2000).

[24] Boyle, E., Gilboa, N., and Ishai, Y. Function secret sharing. In *CRYPTO* (2015), Springer, pp. 337–367.

[25] Boyle, E., Gilboa, N., and Ishai, Y. Function secret sharing: Improvements and extensions. In *CCS* (2016), ACM, pp. 1292–1303.

[26] Braun, B., Feldman, A. J., Ren, Z., Setty, S., Blumberg, A. J., and Walfish, M. Verifying computations with state. In *SOSP* (2013), ACM, pp. 341–357.

[27] Brickell, J., and Shmatikov, V. Efficient anonymity-preserving data collection. In *KDD* (2006), ACM, pp. 76–85.

[28] Broadbent, A., and Tapp, A. Information-theoretic security without an honest majority. In *ASIACRYPT* (2007), Springer, pp. 410–426.

[29] Burkhart, M., Strasser, M., Many, D., and Dimitropoulos, X. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. *USENIX Security* (2010).

[30] Castelluccia, C., Mykletun, E., and Tsudik, G. Efficient aggregation of encrypted data in wireless sensor networks. In *MobiQuitous* (2005), IEEE, pp. 109–117.

[31] Chaum, D. The Dining Cryptographers Problem: Unconditional sender and recipient untraceability. *Journal of Cryptology 1*, 1 (1988), 65–75.

[32] Chaum, D. L. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM 24*, 2 (1981), 84–90.

[33] Chen, R., Akkus, I. E., and Francis, P. SplitX: High-performance private analytics. *SIGCOMM 43*, 4 (2013), 315–326.

[34] Chen, R., Reznichenko, A., Francis, P., and Gehrke, J. Towards statistical queries over distributed private user data. In *NSDI* (2012), pp. 169–182.

[35] Chromium source code. https://chromium.googlesource.com/chromium/src/+/master/tools/metrics/rappor/rappor.xml. Accessed 15 September 2016.

[36] Cormode, G., and Muthukrishnan, S. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms 55*, 1 (2005), 58–75.

[37] Cormode, G., Thaler, J., and Yi, K. Verifying computations with streaming interactive proofs. *VLDB 5*, 1 (2011), 25–36.

[38] Corrigan-Gibbs, H., Boneh, D., and Mazières, D. Riposte: An anonymous messaging system handling millions of users. In *Security and Privacy* (2015), IEEE, pp. 321–338.

[39] Corrigan-Gibbs, H., and Ford, B. Dissent: accountable anonymous group messaging. In *CCS* (2010), ACM, pp. 340–350.

[40] Corrigan-Gibbs, H., Wolinsky, D. I., and Ford, B. Proactively accountable anonymous messaging in Verdict. In *USENIX Security* (2013), pp. 147–162.

[41] Costello, C., Fournet, C., Howell, J., Kohlweiss, M., Kreuter, B., Naehrig, M., Parno, B., and Zahur, S. Geppetto: Versatile verifiable computation. In *Security and Privacy* (2015), IEEE, pp. 253–270.

[42] CPP. California Psychological Inventory. https://www.cpp.com/products/cpi/index.aspx. Accessed 15 September 2016.

[43] Cramer, R., and Shoup, V. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *CRYPTO* (1998), Springer, pp. 13–25.

[44] Damgård, I., Keller, M., Larraia, E., Miles, C., and Smart, N. P. Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In *SCN* (2012), Springer, pp. 241–263.

[45] Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., and Smart, N. P. Practical covertly secure MPC for dishonest majority–or: Breaking the SPDZ limits. In *ESORICS* (2013), Springer, pp. 1–18.

[46] Damgård, I., Pastro, V., Smart, N. P., and Zakarias, S. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO* (2012), pp. 643–662.

[47] Danezis, G. *Better anonymous communications*. PhD thesis, University of Cambridge, 2004.

[48] Danezis, G., Fournet, C., Kohlweiss, M., and Zanella-Béguelin, S. Smart meter aggregation via secret-sharing. In *Workshop on Smart Energy Grid Security* (2013), ACM, pp. 75–80.

[49] Danezis, G., and Serjantov, A. Statistical disclosure or intersection attacks on anonymity systems. In *Information Hiding* (2004), Springer, pp. 293–308.

[50] DEDIS Research Lab at EPFL. Advanced crypto library for the Go language. https://github.com/dedis/crypto.

[51] Dingledine, R., Mathewson, N., and Syverson, P. Tor: the second-generation onion router. In *USENIX Security* (2004), USENIX Association, pp. 21–21.

[52] Douceur, J. R. The Sybil Attack. In *International Workshop on Peer-to-Peer Systems* (2002), Springer, pp. 251–260.

[53] Duan, Y., Canny, J., and Zhan, J. P4P: practical large-scale privacy-preserving distributed computation robust against malicious users. In *USENIX Security* (Aug. 2010).

[54] Dwork, C. Differential privacy. In *ICALP* (2006), pp. 1–12.

[55] Dwork, C., Kenthapadi, K., McSherry, F., Mironov, I., and Naor, M. Our data, ourselves: Privacy via distributed noise generation. In *EUROCRYPT*. Springer, 2006, pp. 486–503.

[56] Elahi, T., Danezis, G., and Goldberg, I. PrivEx: Private collection of traffic statistics for anonymous communication networks. In *CCS* (2014), ACM, pp. 1068–1079.

[57] Erlingsson, Ú., Pihur, V., and Korolova, A. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *CCS* (2014), ACM, pp. 1054–1067.

[58] Fanti, G., Pihur, V., and Erlingsson, Ú. Building a RAPPOR with the unknown: Privacy-preserving learning of associations and data dictionaries. *PoPETS 2016*, 3 (2016), 41–61.

[59] FLINT: Fast Library for Number Theory. http://www.flintlib.org/.

[60] Fortnow, L., Rompel, J., and Sipser, M. On the power of multi-prover interactive protocols. *Theoretical Computer Science 134*, 2 (1994), 545–557.

[61] Freedman, M. J., and Morris, R. Tarzan: A peer-to-peer anonymizing network layer. In *CCS* (2002), ACM, pp. 193–206.

[62] Gennaro, R., Gentry, C., Parno, B., and Raykova, M. Quadratic span programs and succinct NIZKs without PCPs. In *CRYPTO* (2013), pp. 626–645.

[63] Gerard, A. B. Parent-Child Relationship Inventory. http://www.wpspublish.com/store/p/2898/parent-child-relationship-inventory-pcri. Accessed 15 September 2016.

[64] Gilboa, N., and Ishai, Y. Distributed point functions and their applications. In *EUROCRYPT* (2014), Springer, pp. 640–658.

[65] Glanz, J., Larson, J., and Lehren, A. W. Spy agencies tap data streaming from phone apps. http://www.nytimes.com/2014/01/28/world/spy-agencies-scour-phone-apps-for-personal-data.html, Jan. 27, 2014. Accessed 20 September 2016.

[66] Goldreich, O. *Foundations of Cryptography*. Cambridge University Press, 2001.

[67] Goldreich, O., Goldwasser, S., and Halevi, S. Collision-free hashing from lattice problems. Cryptology ePrint Archive, Report 1996/009, 1996. http://eprint.iacr.org/1996/009.

[68] Goldreich, O., Micali, S., and Wigderson, A. How to play any mental game. In *STOC* (1987), ACM, pp. 218–229.

[69] Goldwasser, S., Kalai, Y. T., and Rothblum, G. N. Delegating computation: Interactive proofs for Muggles. *Journal of the ACM 62*, 4 (2015), 27.

[70] Goldwasser, S., Micali, S., and Rackoff, C. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing 18*, 1 (1989), 186–208.

[71] Goldwasser, S., Micali, S., and Rivest, R. L. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing 17*, 2 (1988), 281–308.

[72] Greenberg, A. Apple's 'differential privacy' is about collecting your data—but not your data. https://www.wired.com/2016/06/apples-differential-privacy-collecting-data/, June 13, 2016. Accessed 21 September 2016.

[73] Gueron, S., Lindell, Y., Nof, A., and Pinkas, B. Fast garbling of circuits under standard assumptions. In *CCS* (2015), ACM, pp. 567–578.

[74] Guha, S., and McGregor, A. Stream order and order statistics: Quantile estimation in random-order streams. *SIAM Journal on Computing 38*, 5 (2009), 2044–2059.

[75] Hilts, A., Parsons, C., and Knockel, J. Every step you fake: A comparative analysis of fitness tracker privacy and security. Tech. rep., Open Effect, 2016. Accessed 16 September 2016.

[76] Hohenberger, S., Myers, S., Pass, R., and shelat, a. ANONIZE: A large-scale anonymous survey system. In *Security and Privacy* (2014), IEEE, pp. 375–389.

[77] Impagliazzo, R., and Naor, M. Efficient cryptographic schemes provably as secure as subset sum. *Journal of Cryptology 9*, 4 (1996), 199–216.

[78] Janosi, A., Steinbrunn, W., Pfisterer, M., and Detrano, R. Heart disease data set. https://archive.ics.uci.edu/ml/datasets/Heart+Disease, 22 July 1988. Accessed 15 September 2016.

[79] Jawurek, M., and Kerschbaum, F. Fault-tolerant privacy-preserving statistics. In *PETS* (2012), Springer, pp. 221–238.

[80] Jeske, T. Floating car data from smartphones: What Google and Waze know about you and how hackers can control traffic. *BlackHat Europe* (2013).

[81] Joye, M., and Libert, B. A scalable scheme for privacy-preserving aggregation of time-series data. In *Financial Cryptography* (2013), pp. 111–125.

[82] Karr, A. F., Lin, X., Sanil, A. P., and Reiter, J. P. Secure regression on distributed databases. *Journal of Computational and Graphical Statistics 14*, 2 (2005), 263–279.

[83] Kedogan, D., Agrawal, D., and Penz, S. Limits of anonymity in open environments. In *Information Hiding* (2002), Springer, pp. 53–69.

[84] Keller, J., Lai, K. R., and Perlroth, N. How many times has your personal information been exposed to hackers? http://www.nytimes.com/interactive/2015/07/29/technology/personaltech/what-parts-of-your-information-have-been-exposed-to-hackers-quiz.html, July 29, 2015.

[85] Krawczyk, H. Secret sharing made short. In *CRYPTO* (1993), pp. 136–146.

[86] Kursawe, K., Danezis, G., and Kohlweiss, M. Privacy-friendly aggregation for the smart-grid. In *PETS* (2011), pp. 175–191.

[87] Kwon, A., Lazar, D., Devadas, S., and Ford, B. Riffle. *Proceedings on Privacy Enhancing Technologies 2016*, 2 (2015), 115–134.

[88] Le Blond, S., Choffnes, D., Zhou, W., Druschel, P., Ballani, H., and Francis, P. Towards efficient traffic-analysis resistant anonymity networks. In *SIGCOMM* (2013), ACM.

[89] Lindell, Y. Fast cut-and-choose-based protocols for malicious and covert adversaries. *Journal of Cryptology 29*, 2 (2016), 456–490.

[90] Lindell, Y., and Pinkas, B. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology 22*, 2 (2009), 161–188.

[91] Malkhi, D., Nisan, N., Pinkas, B., Sella, Y., et al. Fairplay–secure two-party computation system. In *USENIX Security* (2004).

[92] Melis, L., Danezis, G., and De Cristofaro, E. Efficient private statistics with succinct sketches. In *NDSS* (Feb. 2016), Internet Society.

[93] Mohassel, P., Rosulek, M., and Zhang, Y. Fast and secure three-party computation: The Garbled Circuit approach. In *CCS* (2015), ACM, pp. 591–602.

[94] Murdoch, S. J., and Danezis, G. Low-cost traffic analysis of Tor. In *Security and Privacy* (2005), IEEE, pp. 183–195.

[95] Neff, C. A. A verifiable secret shuffle and its application to e-voting. In *CCS* (2001), ACM, pp. 116–125.

[96] Nikolaenko, V., Ioannidis, S., Weinsberg, U., Joye, M., Taft, N., and Boneh, D. Privacy-preserving matrix factorization. In *CCS* (2013), ACM, pp. 801–812.

[97] Parno, B., Howell, J., Gentry, C., and Raykova, M. Pinocchio: Nearly practical verifiable computation. In *Security and Privacy* (2013), IEEE, pp. 238–252.

[98] Pinkas, B., Schneider, T., Smart, N. P., and Williams, S. C. Secure two-party computation is practical. In *CRYPTO* (2009), Springer, pp. 250–267.

[99] Popa, R. A., Balakrishnan, H., and Blumberg, A. J. VPriv: Protecting privacy in location-based vehicular services. In *USENIX Security* (2009), pp. 335–350.

[100] Popa, R. A., Blumberg, A. J., Balakrishnan, H., and Li, F. H. Privacy and accountability for location-based aggregate statistics. In *CCS* (2011), ACM, pp. 653–666.

[101] Reiter, M. K., and Rubin, A. D. Crowds: Anonymity for web transactions. *TISSEC 1*, 1 (1998), 66–92.

[102] Rogaway, P., and Bellare, M. Robust computational secret sharing and a unified account of classical secret-sharing goals. In *CCS* (2007), ACM, pp. 172–184.

[103] Schnorr, C.-P. Efficient signature generation by smart cards. *Journal of Cryptology 4*, 3 (1991), 161–174.

[104] Schwartz, J. T. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM 27*, 4 (1980), 701–717.

[105] Serjantov, A., Dingledine, R., and Syverson, P. From a trickle to a flood: Active attacks on several mix types. In *Information Hiding* (2002), Springer, pp. 36–52.

[106] Shamir, A. How to share a secret. *Communications of the ACM 22*, 11 (1979), 612–613.

[107] Shi, E., Chan, T. H., Rieffel, E., Chow, R., and Song, D. Privacy-preserving aggregation of time-series data. In *NDSS* (2011), vol. 2, Internet Society, pp. 1–17.

[108] Shoup, V. OAEP reconsidered. In *CRYPTO* (2001), Springer, pp. 239–259.

[109] Shoup, V. A proposal for an ISO standard for public key encryption (version 2.1). *Cryptology ePrint Archive, Report 2001/112* (2001). http://eprint.iacr.org/2001/112.

[110] Sirer, E. G., Goel, S., Robson, M., and Engin, D. Eluding carnivores: File sharing with strong anonymity. In *ACM SIGOPS European Workshop* (2004), ACM, p. 19.

[111] Smart, N. FHE-MPC notes. https://www.cs.bris.ac.uk/~nigel/FHE-MPC/Lecture8.pdf, Nov. 2011. Scribed by Peter Scholl.

[112] Smith, B. Uber executive suggests digging up dirt on journalists. https://www.buzzfeed.com/bensmith/uber-executive-suggests-digging-up-dirt-on-journalists, Nov. 17, 2014. Accessed 20 September 2016.

[113] U.S. Department of Health and Human Services. AIDSinfo. https://aidsinfo.nih.gov/apps.

[114] Viswanath, B., Post, A., Gummadi, K. P., and Mislove, A. An analysis of social network-based Sybil defenses. *SIGCOMM 40*, 4 (2010), 363–374.

[115] Wahby, R. S., Setty, S. T., Ren, Z., Blumberg, A. J., and Walfish, M. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS* (2016).

[116] Walfish, M., and Blumberg, A. J. Verifying computations without reexecuting them. *Communications of the ACM 58*, 2 (2015), 74–84.

[117] Wang, G., Wang, B., Wang, T., Nika, A., Zheng, H., and Zhao, B. Y. Defending against Sybil devices in crowdsourced mapping services. In *MobiSys* (2016), ACM, pp. 179–191.

[118] Warner, S. L. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association 60*, 309 (1965), 63–69.

[119] Williams, R. R. Strong ETH breaks with Merlin and Arthur: Short non-interactive proofs of batch evaluation. In *31st Conference on Computational Complexity* (2016).

[120] Wolberg, W. H., Street, W. N., and Mangasarian, O. L. Wisconsin prognostic breast cancer data set. https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/, Dec. 1995. Accessed 15 September 2016.

[121] Wolinsky, D. I., Corrigan-Gibbs, H., Ford, B., and Johnson, A. Dissent in numbers: Making strong anonymity scale. In *OSDI* (2012), pp. 179–182.

[122] Wolinsky, D. I., Syta, E., and Ford, B. Hang with your buddies to resist intersection attacks. In *CCS* (2013), ACM, pp. 1153–1166.

[123] Yao, A. C.-C. How to generate and exchange secrets. In *FOCS* (1986), IEEE, pp. 162–167.

[124] Yu, H., Gibbons, P. B., Kaminsky, M., and Xiao, F. SybilLimit: A near-optimal social network defense against Sybil attacks. In *Security and Privacy* (2008), IEEE, pp. 3–17.

[125] Yu, H., Kaminsky, M., Gibbons, P. B., and Flaxman, A. SybilGuard: defending against Sybil attacks via social networks. In *SIGCOMM* (2006), vol. 36, ACM, pp. 267–278.

[126] Zippel, R. Probabilistic algorithms for sparse polynomials. In *Proceedings of the International Symposiumon on Symbolic and Algebraic Computation* (1979), Springer-Verlag, pp. 216–226.

A Security definitions

We use the standard definitions of *negligible functions* and *computational indistinguishability*. Goldreich [66] gives a formal treatment of these concepts. For clarity, we often prefer to leave the security parameter implicit.

It is possible to make our notion of privacy formal with a simulation-based definition. The following informal definition captures the essence of the full formalism:

Definition 1 (f-Privacy)**.** Say that there are s servers and n clients in a Prio deployment. We say that the scheme provides *f-privacy* for a function f, if for:

- every subset of at most $s-1$ servers, and
- every subset of at most n clients,

there exists an efficient simulator that, for every choice of client inputs $(x_1, \ldots, x_n)$, takes as input:

- the public parameters to the protocol run (all participants' public keys, the description of the aggregation function f, the cryptographic parameters, etc.),
- the indices of the adversarial clients and servers,
- oracle access to the adversarial participants, and
- the value $f(x_1, \ldots, x_n)$,

and outputs a simulation of the adversarial participants' view of the protocol run whose distribution is computationally indistinguishable from the distribution of the adversary's view of the real protocol run.

Let SORT be the function that takes n inputs and outputs them in lexicographically increasing order.

Definition 2 (Anonymity)**.** We say that a data-collection scheme provides *anonymity* if it provides f-privacy, in the sense of Definition 1, for $f = \mathsf{SORT}$.

A scheme that provides this form of anonymity leaks to the adversary the entire list of client inputs $(x_1, \ldots, x_n)$, but the adversary learns nothing about which client submitted which value x_i. For example, if each client submits their location via a data-collection scheme that provides anonymity, the servers learn the list of submitted locations $\{\ell_1, \ldots, \ell_n\}$, but the servers learn nothing about whether client x or y is in a particular location ℓ^*.

Definition 3. A function $f(x_1, \ldots, x_n)$ is *symmetric* if, for all permutations π on n elements, the equality $f(x_1, \ldots, x_n) = f(x_{\pi(1)}, \ldots, x_{\pi(n)})$ holds.

Claim 4. *Let $\mathcal{D}$ be a data-collection scheme that provides f-privacy, in the sense of Definition 1, for a symmetric function f. Then $\mathcal{D}$ provides anonymity.*

Proof sketch. The fact that $\mathcal{D}$ provides f-privacy implies the existence of a simulator $S_{\mathcal{D}}$ that takes as input $f(x_1, \ldots, x_n)$, along with other public values, and induces a distribution of protocol transcripts indistinguishable from the real one. If f is symmetric, $f(x_1, \ldots, x_n) = f(x'_1, \ldots, x'_n)$, where

$$(x'_1, \ldots, x'_n) = \mathsf{SORT}(x_1, \ldots, x_n).$$

Using this fact, we construct the simulator required for the anonymity definition: on input $(x'_1, \ldots, x'_n) = \mathsf{SORT}(x_1, \ldots, x_n)$, compute $f(x'_1, \ldots, x'_n)$, and feed the output of f to the simulator $S_{\mathcal{D}}$. The validity of the simulation is immediate. □

The following claim demonstrates that it really only makes sense to use an f-private data collection scheme when the function f is symmetric, as all of the functions we consider in Prio are.

Claim 5. *Let f be a non-symmetric function. Then there is no anonymous data collection scheme that correctly computes f.*

Proof sketch. Because f is not symmetric, there exists an input $(x_1, \ldots, x_n)$ in the domain of f, and a permutation π on n elements, such that $f(x_1, \ldots, x_n) \neq f(x_{\pi(1)}, \ldots, x_{\pi(n)})$.

Let $\mathcal{D}$ be a data-collection scheme that implements the aggregation function $f(x_1, \ldots, x_n)$. This $\mathcal{D}$ outputs $f(x_1, \ldots, x_n)$ for all $x_1, \ldots, x_n$ in the domain, and hence $f(x_1, \ldots, x_n)$ is part of the protocol transcript.

For $\mathcal{D}$ to be anonymous, there must be a simulator that takes $\mathsf{SORT}(x_1, \ldots, x_n)$ as input, and simulates the protocol transcript. In particular, it must output $f(x_1, \ldots, x_n)$. But given $\mathsf{SORT}(x_1, \ldots, x_n)$, it will necessarily fail to output the correct protocol transcript on either $(x_1, \ldots, x_n)$ or $(x_{\pi(1)}, \ldots, x_{\pi(n)})$. □

Robustness. Recall that each Prio client holds a value x_i, where the value x_i is an element of some set of data items $\mathcal{D}$. For example, $\mathcal{D}$ might be the set of 4-bit integers. The definition of robustness states that when all servers are honest, a set of malicious clients cannot influence the final aggregate, beyond their ability to choose arbitrary *valid* inputs. For example, malicious clients can choose arbitrary 4-bit integers as their input values, but cannot influence the output in any other way.

Definition 6 (Robustness)**.** Fix a security parameter $\lambda > 0$. We say that an n-client Prio deployment provides *robustness* if, when all Prio servers execute the protocol faithfully, for every number m of malicious clients (with $0 \leq m \leq n$), and for every choice of honest client's inputs $(x_1, \ldots, x_{n-m}) \in \mathcal{D}^{n-m}$, the servers, with all but negligible probability in λ, output a value in the set:

$$\left\{ f(x_1, \ldots, x_n) \mid (x_{n-m+1}, \ldots, x_n) \in \mathcal{D}^m \right\}.$$

B Robustness against faulty servers

If at least one of the servers is honest, Prio ensures that the adversary learns nothing about clients' data, except the aggregate statistic. However, Prio provides *robustness* only if all servers are honest.

Providing robustness in the face of faulty servers is obviously desirable, but we are not convinced that it is worth the security and performance costs. First, providing robustness necessarily weakens the privacy guarantees that the system provides: if the system protects *robustness* in the presence of k faulty servers, then the system can protect *privacy* only against a coalition of at most $s - k - 1$ malicious servers. The reason is that, if robustness holds against k faulty servers, then $s - k$ honest servers must be able to produce a correct output even if these k faulty

servers are offline. Put another way: $s - k$ *dishonest* servers can recover the output of the system even without the participation of the k honest servers. Instead of computing an aggregate over many clients ($f(x_1, \ldots, x_n)$), the dishonest servers can compute the "aggregate" over a single client's submission ($f(x_1)$) and essentially learn that client's private data value.

So strengthening robustness in this setting weakens privacy. Second, protecting robustness comes at a performance cost: some our optimizations use a "leader" server to coordinate the processing of each client submission (see Appendix H). A faulty leader cannot compromise privacy, but *can* compromise robustness. Strengthening the robustness property would force us to abandon these optimizations.

That said, it would be possible to extend Prio to provide robustness in the presence of corrupt servers using standard techniques [10] (replace s-out-of-s secret sharing with Shamir's threshold secret-sharing scheme [106], etc.).

C MPC background

This appendix reviews the definition of arithmetic circuits and Donald Beaver's multi-party computation protocol [9].

C.1 Definition: arithmetic circuits

An *arithmetic circuit* $\mathcal{C}$ over a finite field $\mathbb{F}$ takes as input a vector $x = \langle x^{(1)}, \ldots, x^{(L)} \rangle \in \mathbb{F}^L$ and produces a single field element as output. We represent the circuit as a directed acyclic graph, in which each vertex in the graph is either an *input*, a *gate*, or an *output* vertex.

Input vertices have in-degree zero and are labeled with a variable in $\{x^{(1)}, \ldots, x^{(L)}\}$ or a constant in $\mathbb{F}$. Gate vertices have in-degree two and are labeled with the operation $+$ or $\times$. The circuit has a single output vertex, which has out-degree zero.

To compute the circuit $\mathcal{C}(x) = \mathcal{C}(x^{(1)}, \ldots, x^{(L)})$, we walk through the circuit from inputs to outputs, assigning a value in $\mathbb{F}$ to each wire until we have a value on the output wire, which is the value of $\mathcal{C}(x)$. In this way, the circuit implements a mapping $\mathcal{C} : \mathbb{F}^L \to \mathbb{F}$.

C.2 Beaver's MPC protocol

This discussion draws on the clear exposition by Smart [111].

Each server starts the protocol holding a share $[x]_i$ of an input vector x. The servers want to compute $\mathcal{C}(x)$, for some arithmetic circuit $\mathcal{C}$.

The multi-party computation protocol walks through the circuit $\mathcal{C}$ wire by wire, from inputs to outputs. The protocol maintains the invariant that, at the t-th time step, each server holds a share of the value on the t-th wire in the circuit. At the first step, the servers hold shares of the input wires (by construction) and in the last step of the protocol, the servers hold shares of the output wire. The servers can then publish their shares of the output wires, which allows them all to reconstruct the value of $\mathcal{C}(x)$. To preserve privacy, no subset of the servers must ever have enough information to recover the value on any internal wire in the circuit.

There are only two types of gates in an arithmetic circuit (addition gates and multiplication gates), so we just have to show how the servers can compute the shares of the outputs of these gates from shares of the inputs. All arithmetic in this section is in a finite field $\mathbb{F}$.

Addition gates. In the computation of an addition gate "$y + z$", the ith server holds shares $[y]_i$ and $[z]_i$ of the input wires and the server needs to compute a share of $y + z$. To do so, the server can just add its shares locally

$$[y + z]_i = [y]_i + [z]_i.$$

Multiplication gates. In the computation of a multiplication gate, the ith server holds shares $[y]_i$ and $[z]_i$ and wants to compute a share of yz.

When one of the inputs to a multiplication gate is a constant, each server can locally compute a share of the output of the gate. For example, to multiply a share $[y]_i$ by a constant $A \in \mathbb{F}$, each server i computes their share of the product as $[Ay]_i = A[y_i]$.

Beaver showed that the servers can use pre-computed *multiplication triples* to evaluate multiplication gates [9]. A multiplication triple is a one-time-use triple of values $(a, b, c) \in \mathbb{F}^3$, chosen at random subject to the constraint that $a \cdot b = c \in \mathbb{F}$. When used in the context of multi-party computation, each server i holds a share $([a]_i, [b]_i, [c]_i) \in \mathbb{F}^3$ of the triple.

Using their shares of one such triple (a, b, c), the servers can jointly evaluate shares of the output of a multiplication gate yz. To do so, each server i uses her shares $[y]_i$ and $[z]_i$ of the input wires, along with the first two components of its multiplication triple to compute the following values:

$$[d]_i = [y]_i - [a]_i \quad ; \quad [e]_i = [z]_i - [b]_i.$$

Each server i then broadcasts $[d]_i$ and $[e]_i$. Using the broadcasted shares, every server can reconstruct d and e and can compute:

$$\sigma_i = de/s + d[b]_i + e[a]_i + [c]_i.$$

Recall that s is the number of servers—a public constant—and the division symbol here indicates division (i.e., inversion then multiplication) in the field $\mathbb{F}$. A few lines of

arithmetic confirm that σ_i is a sharing of the product yz. To prove this we compute:

$$\begin{aligned}\sum_i \sigma_i &= \sum_i (de/s + d[b]_i + e[a]_i + [c]_i) \\ &= de + db + ea + c \\ &= (y-a)(z-b) + (y-a)b + (z-b)a + c \\ &= (y-a)z + (z-b)a + c \\ &= yz - az + az - ab + c \\ &= yz - ab + c \\ &= yz.\end{aligned}$$

The last step used that $c = ab$ (by construction of the multiplication triple), so: $\sum_i \sigma_i = yz$, which implies that $\sigma_i = [yz]_i$.

Since the servers can perform addition and multiplication of shared values, they can compute any function of the client's data value in this way, as long as they have a way of securely generating multiplication triples. The expensive part of traditional MPC protocols is the process by which mutually distrusting servers generate these triples in a distributed way.

D Server-side Valid computation

If the Valid predicate takes secret inputs from the servers, the servers can compute $\mathsf{Valid}(x)$ on a client-provided input x without learning anything about x, except the value of $\mathsf{Valid}(x)$. In addition, the client learns nothing about the Valid circuit, except the number of multiplication gates in the circuit.

Let M be the number of multiplication gates in the Valid circuit. To execute the Valid computation on the server side, the client sends M multiplication triple shares (defined in Appendix C.2) to each server, along with a share of its private value x. Let the t-th multiplication triple be of the form $(a_t, b_t, c_t) \in \mathbb{F}^3$. Then define a circuit $\mathcal{M}$ that returns "1" if and only if $c_t = a_t \cdot b_t$, for all $1 \le t \le M$.

The client can use a SNIP proof (Section 4.1) to convince the servers that all of the M triples it sent the servers are well-formed. Then, the servers can execute Beaver's multiparty computation protocol (Section C.2) to evaluate the circuit using the M client-provided multiplication triples.

Running the computation requires the servers to exchange $\Theta(M)$ field elements, and the number of rounds of communication is proportional to the multiplicative depth of the Valid circuit (i.e., the maximum number of multiplication gates on an input-output path).

E AFE definitions

An AFE is defined relative to a field $\mathbb{F}$, two integers k and k' (where $k' \le k$), a set $\mathcal{D}$ of data elements, a set $\mathcal{A}$ of possible values of the aggregate statistic, and an aggregation function $f : \mathcal{D}^n \to \mathcal{A}$. An AFE scheme consists of three efficient algorithms. The algorithms are:

- $\mathsf{Encode} : \mathcal{D} \to \mathbb{F}^k$. Covert a data item into its AFE-encoded counterpart.
- $\mathsf{Valid} : \mathbb{F}^k \to \{0, 1\}$. Return "1" if and only if the input is in the image of Encode.
- $\mathsf{Decode} : \mathbb{F}^{k'} \to \mathcal{A}$. Given a vector representing a collection of encoded data items, return the value of the aggregation function f evaluated at these items.

To be useful, an AFE encoding should satisfy the following properties:

Definition 7 (AFE correctness)**.** We say that an AFE is *correct* for an aggregation function f if, for every choice of $(x_1, \ldots, x_n) \in \mathcal{D}^n$, we have that:

$$\mathsf{Decode}(\textstyle\sum_i \mathsf{Trunc}_{k'}(\mathsf{Encode}(x_i))) = f(x_1, \ldots, x_n).$$

Recall that $\mathsf{Trunc}_{k'}(v)$ denotes truncating the vector $v \in \mathbb{F}_p^k$ to its first k' components.

The correctness property of an AFE essentially states that if we are given valid encodings of data items $(x_1, \ldots, x_n) \in \mathcal{D}^n$, the decoding of their sum should be $f(x_1, \ldots, x_n)$.

Definition 8 (AFE soundness)**.** We say that an AFE is *sound* if, for all encodings $e \in \mathbb{F}^k$: the predicate $\mathsf{Valid}(e) = 1$ if and only if there exists a data item $x \in \mathcal{D}$ such that $e = \mathsf{Encode}(x)$.

An AFE is private with respect to a function $\hat{f}$, if the sum of encodings $\sigma = \sum_i \mathsf{Trunc}_{k'}(\mathsf{Encode}(x_i))$, given as input to algorithm Decode, reveals nothing about the underlying data beyond what $\hat{f}(x_1, \ldots, x_n)$ reveals.

Definition 9 (AFE privacy)**.** We say that an AFE is *private* with respect to a function $\hat{f} : \mathcal{D}^n \to \mathcal{A}'$ if there exists an efficient simulator S such that for all input data $(x_1, \ldots, x_n) \in \mathcal{D}^n$, the distribution $S(\hat{f}(x_1, \ldots, x_n))$ is indistinguishable from the distribution $\sigma = \sum_i \mathsf{Trunc}_{k'}(\mathsf{Encode}(x_i))$.

Relaxed correctness. In many cases, randomized data structures are more efficient than their deterministic counterparts. We can define a relaxed notion of correctness to capture a correctness notion for randomized AFEs. In the randomized case, the scheme is parameterized by constants $0 < \delta, \epsilon$ and the Decode algorithm may use randomness. We demand that with probability at least

$1 - 2^{-\delta}$, over the randomness of the algorithms, the encoding yields an "ϵ-good" approximation of f. In our applications, typically an ϵ-good approximation is within a multiplicative or additive factor of ϵ from the true value; the exact meaning depends on the AFE in question.

F Additional AFEs

Approximate counts. The frequency count AFE, presented in Section 5.2, works well when the client value x lies in a small set of possible data values $\mathcal{D}$. This AFE requires communication linear in the size of $\mathcal{D}$. When the set $\mathcal{D}$ is large, a more efficient solution is to use a randomized counting data structure, such as a count-min sketch [36].

Melis et al. [92] demonstrated how to combine a count-min sketch with a secret-sharing scheme to efficiently compute counts over private data. We can make their approach robust to malicious clients by implementing a count-min sketch AFE in Prio. To do so, we use $\ln(1/\delta)$ instances of the basic frequency count AFE, each for a set of size e/ϵ, for some constants ϵ and δ, and where $e \approx 2.718$. With n client inputs, the count-min sketch yields counts that are at most an additive ϵn overestimate of the true values, except with probability $e^{-\delta}$.

Crucially, the Valid algorithm for this composed construction requires a relatively small number of multiplication gates—a few hundreds, for realistic choices of ϵ and δ—so the servers can check the correctness of the encodings efficiently.

This AFE leaks the contents of a count-min sketch data structure into which all of the clients' values $(x_1, \ldots, x_n)$ have been inserted.

Share compression. The output of the count-min sketch AFE encoding routine is essentially a very sparse matrix of dimension $\ln(1/\delta) \times (e/\epsilon)$. The matrix is all zeros, except for a single "1" in each row. If the Prio client uses a conventional secret-sharing scheme to split this encoded matrix into s shares—one per server—the size of each share would be as large as the matrix itself, even though the plaintext matrix contents are highly compressible.

A more efficient way to split the matrix into shares would be to use a function secret-sharing scheme [24, 25, 64]. Applying a function secret sharing scheme to each row of the encoded matrix would allow the size of each share to grow as the square-root of the matrix width (instead of linearly). When using Prio with only two servers, there are very efficient function secret-sharing constructions that would allow the shares to have length logarithmic in the width of the matrix [25]. We leave further exploration of this technique to future work.

Most popular. Another common task is to return the most popular string in a data set, such as the most popular homepage amongst a set of Web clients. When the universe of strings is small, it is possible to find the most popular string using the frequency-counting AFE. When the universe is large (e.g., the set of all URLs), this method is not useful, since recovering the most popular string would require querying the structure for the count of every possible string. Instead, we use a simplified version of a data structure of Bassily and Smith [7].

When there is a very popular string—one that more than $n/2$ clients hold, we can construct a very efficient AFE for collecting it. Let $\mathbb{F}$ be a field of size at least n. The $\mathsf{Encode}(x)$ algorithm represents its input x as a b-bit string $x = (x_0, x_1, x_2, \ldots, x_{b-1}) \in \{0,1\}^b$, and outputs a vector of b field elements $(\beta_0, \ldots, \beta_{b-1}) \in \mathbb{F}^b$, where $\beta_i = x_i$ for all i. The Valid algorithm uses b multiplication gates to check that each value β_i is really a 0/1 value in $\mathbb{F}$, as in the summation AFE.

The Decode algorithm gets as input the sum of n such encodings $\sigma = \sum_{i=1}^{n} \mathsf{Encode}(x_i) = (e_0, \ldots, e_{b-1}) \in \mathbb{F}^b$. The Decode algorithm rounds each value e_i either down to zero or up to n (whichever is closer) and then normalizes the rounded number by n to get a b-bit binary string $\sigma \in \{0,1\}^b$, which it outputs. As long as there is a string σ^* with popularity greater than 50%, this AFE returns it. To see why, consider the first bit of σ. If $\sigma^*[0] = 0$, then the sum $e_0 < n/2$ and Decode outputs "0." If $\sigma^*[0] = 1$, then the sum $e_0 > n/2$ and Decode outputs "1." Correctness for longer strings follows

This AFE leaks quite a bit of information about the given data. Given σ, one learns the number of data values that have their ith bit set to 1, for every $0 \le i < b$. In fact, the AFE is private relative to a function that outputs these b values, which shows that nothing else is leaked by σ.

With a significantly more complicated construction, we can adapt a similar idea to collect strings that a constant fraction c of clients hold, for $c \le 1/2$. The idea is to have the servers drop client-submitted strings at random into different "buckets," such that at least one bucket has a very popular string with high probability [7].

Evaluating an arbitrary ML model. We wish to measure how well a public regression model predicts a target y from a client-submitted feature vector x. In particular, if our model outputs a prediction $\hat{y} = M(x)$, we would like to measure how good of an approximation $\hat{y}$ is of y. The R^2 coefficient is one statistic for capturing this information.

Karr et al. [82] observe that it is possible to reduce the problem of computing the R^2 coefficient of a public regression model to the problem of computing private sums. We can adopt a variant of this idea to use Prio to compute the R^2 coefficient in a way that leaks little beyond the coefficient itself.

The R^2-coefficient of the model for client inputs $\{x_1, \ldots, x_n\}$ is $R^2 = 1 - \sum_{i=1}^{2} (y_i - \hat{y}_i)^2 / \mathrm{Var}(y_1, \ldots, y_n)$,

where y_i is the true value associated with x_i, $\hat{y}_i = M(x_i)$ is the predicted value of y_i, and $\mathrm{Var}(\cdot)$ denotes variance.

An AFE for computing the R^2 coefficient works as follows. On input (x, y), the Encode algorithm first computes the prediction $\hat{y} = M(x)$ using the public model M. The Encode algorithm then outputs the tuple $(y, y^2, (y-\hat{y})^2, x)$, embedded in a finite field large enough to avoid overflow.

Given the tuple (y, Y, Y^*, x) as input, the Valid algorithm ensures that $Y = y^2$ and $Y^* = (y - M(x))^2$. When the model M is a linear regression model, algorithm Valid can be represented as an arithmetic circuit that requires only two multiplications. If needed, we can augment this with a check that the x values are integers in the appropriate range using a range check, as in prior AFEs. Finally, given the sum of encodings restricted to the first three components, the Decode algorithm has the information it needs to compute the R^2 coefficient.

This AFE is private with respect to a function that outputs the R^2 coefficient, along with the expectation and variance of $\{y_1, \ldots, y_n\}$.

G Prio protocol and proof sketch

We briefly review the full Prio protocol and then discuss its security.

The final protocol. We first review the Prio protocol from Section 5. We assume that every client i, for $i \in \{1, \ldots, n\}$, holds a private value x_i that lies in some set of data items $\mathcal{D}$. We want to compute an aggregation function $f : \mathcal{D}^n \to \mathcal{A}$ on these private values using an AFE. The AFE encoding algorithm Encode maps $\mathcal{D}$ to $\mathbb{F}^k$, for some field $\mathbb{F}$ and an arity k. When decoding, the encoded vectors in $\mathbb{F}^k$ are first truncated to their first k' components.

The Prio protocol proceeds in four steps:

1. **Upload.** Each client i computes $y_i \leftarrow \mathsf{Encode}(x_i)$ and splits its encoded value into s shares, one per server. To do so, the client picks random values $[y_i]_1, \ldots, [y_i]_s \in \mathbb{F}^k$, subject to the constraint: $y_i = [y_i]_1 + \cdots + [y_i]_s \in \mathbb{F}^k$. The client then sends, over an encrypted and authenticated channel, one share of its submission to each server, along with a share of a SNIP proof (Section 4) that $\mathsf{Valid}(y_i) = 1$.

2. **Validate.** Upon receiving the ith client submission, the servers verify the client-provided SNIP to jointly confirm that $\mathsf{Valid}(y_i) = 1$ (i.e., that client's submission is well-formed). If this check fails, the servers reject the submission.

3. **Aggregate.** Each server j holds an accumulator value $A_j \in \mathbb{F}^{k'}$, initialized to zero, where $0 < k' \leq k$. Upon receiving a share of a client encoding $[y_i]_j \in \mathbb{F}^k$, the server truncates $[y_i]_j$ to its first k' components, and adds this share to its accumulator:

$$A_j \leftarrow A_j + \mathsf{Trunc}_{k'}([y_i]_j) \in \mathbb{F}^{k'}.$$

   Recall that $\mathsf{Trunc}_{k'}(v)$ denotes truncating the vector $v \in \mathbb{F}_p^k$ to its first k' components.

4. **Publish.** Once the servers have received a share from each client, they publish their accumulator values. The sum of the accumulator values $\sigma = \sum_j A_j \in \mathbb{F}^{k'}$ yields the sum $\sum_i \mathsf{Trunc}_{k'}(y_i)$ of the clients' private encoded values. The servers output $\mathsf{Decode}(\sigma)$.

Security. We briefly sketch the security argument for the complete protocol. The security definitions appear in Appendix A.

First, the robustness property (Definition 6) follows from the soundness of the SNIP construction: as long as the servers are honest, they will correctly identify and reject any client submissions that do not represent proper AFE encodings.

Next, we argue f-privacy (Definition 1). Define the function

$$g(x_1, \ldots, x_n) = \sum_i \mathsf{Trunc}_{k'}(\mathsf{Encode}(x_i)).$$

We claim that, as long as:

- at least one server executes the protocol correctly,
- the AFE construction is private with respect to f, in the sense of Definition 9, and
- the SNIP construction satisfies the zero-knowledge property (Section 4.1),

the only information that leaks to the adversary is the value of the function f on the clients' private values.

To show this, it suffices to construct a simulator S that takes as input $\sigma = g(x_1, \ldots, x_n)$ and outputs a transcript of the protocol execution that is indistinguishable from a real transcript. Recall that the AFE simulator takes $f(x_1, \ldots, x_n)$ as input and simulates σ. Composing the simulator S with the AFE simulator yields a simulator for the entire protocol, as required by Definition 1.

On input σ, the simulator S executes these steps:

- To simulate the submitted share $[x]_i$ of an honest client, the simulator samples a vector of random field elements of the appropriate length.
- To simulate the SNIP of an honest client, the simulator invokes the SNIP simulator as a subroutine.
- To simulate the adversarially produced values, the simulator can query the adversary (presented as an oracle) on the honest parties' values generated so far.
- To simulate the values produced by the honest servers in Step 4 of the protocol, the simulator picks random

values A_j subject to the constraints $\sigma = \sum_j A_j$, and such that the A_js for the adversarial servers are consistent with their views of the protocol.

As long as there exists a single honest server that the adversary does not control, the adversary sees at most $s-1$ shares of secret-shared values split into s shares throughout the entire protocol execution. These values are trivial to simulate, since they are indistinguishable from random to the adversary.

Finally, anonymity (Definition 2) follows by Claim 4 whenever the function f is symmetric. Otherwise, anonymity is impossible, by Claim 5.

H Additional optimizations

Optimization: PRG secret sharing. The Prio protocol uses additive secret sharing to split the clients' private data into shares. The naïve way to split a value $x \in \mathbb{F}^L$ into s shares is to choose $[x]_1, \ldots, [x]_{s-1} \in \mathbb{F}^L$ at random and then set $[x]_s = x - \sum_{i=1}^{s-1} [x]_i \in \mathbb{F}^L$. A standard bandwidth-saving optimization is to generate the first $s-1$ shares using a pseudo-random generator (PRG) $G : \mathcal{K} \to \mathbb{F}^L$, such as AES in counter mode [85, 102]. To do so, pick $s-1$ random PRG keys $k_1, \ldots, k_{s-1} \in \mathcal{K}$, and define the first $s-1$ shares as $G(k_1), G(k_2), \ldots, G(k_{s-1})$. Rather than representing the first $s-1$ shares as vectors in $\mathbb{F}^L$, we can now represent each of the first $s-1$ shares using a single AES key. (The last share will still be L field elements in length.) This optimization reduces the total size of the shares from sL field elements down to $L + O(1)$. For $s = 5$ servers, this 5× bandwidth savings is significant.

Optimization: Verification without interpolation. In the course of verifying a SNIP (Section 4.2, Step 2), each server i needs to interpolate two large polynomials $[f]_i$ and $[g]_i$. Then, each server must evaluate the polynomials $[f]_i$ and $[g]_i$ at a randomly chosen point $r \in \mathbb{F}$ to execute the randomized polynomial identity test (Step 3).

The degree of these polynomials is close to M, where M is the number of multiplication gates in the Valid circuit. If the servers used straightforward polynomial interpolation and evaluation to verify the SNIPs, the servers would need to perform $\Theta(M \log M)$ multiplications to process a single client submission, even using optimized FFT methods. When the Valid circuit is complex (i.e., $M \approx 2^{16}$ or more), this $\Theta(M \log M)$ cost will be substantial.

Let us imagine for a minute that we could fix *in advance* the random point r that the servers use to execute the polynomial identity test. In this case, each server can perform interpolation and evaluation of any polynomial P in one step using only M field multiplications per server, instead of $\Theta(M \log M)$. To do so, each server precomputes constants $(c_0, \ldots, c_{M-1}) \in \mathbb{F}^M$. These constants depend on the x-coordinates of the points being interpolated (which are always fixed in our application) and on the point r (which for now we assume is fixed). Then, given points $\{(t, y_t)\}_{t=0}^{M-1}$ on a polynomial P, the servers can evaluate P at r using a fast inner-product computation: $P(x) = \sum_t c_t y_t \in \mathbb{F}$. Standard Lagrangian interpolation produces these c_is as intermediate values [5].

Our observation is that the servers *can* fix the "random" point r at which they evaluate the polynomials $[f]_i$ and $[g]_i$ as long as: (1) the clients never learn r, and (2) the servers sample a new random point r periodically. The randomness of the value r only affects soundness. Since we require soundness to hold only if all Prio servers are honest, we may assume that the servers will never reveal the value r to the clients.

A malicious client may try to learn something over time about the servers' secret value r by sending a batch of well-formed and malformed submissions and seeing which submissions the servers do or do not accept. A simple argument shows that after making q such queries, the client's probability of cheating the servers is at most $2Mq/|\mathbb{F}|$. By sampling a new point after every $Q \approx 2^{10}$ client uploads, the servers can amortize the cost of doing the interpolation precomputation over Q client uploads, while keeping the failure probability bounded above by $2MQ/|\mathbb{F}|$, which they might take to be 2^{-60} or less.

In Prio, we apply this optimization to combine the interpolation of $[f]_i$ and $[g]_i$ with the evaluation of these polynomials at the point r.

In Step 2 of the SNIP verification process, each server must also evaluate the client-provided polynomial $[h]_i$ at each point $t \in \{1, \ldots, M\}$. To eliminate this cost, we have the client send the polynomial $[h]_i$ to each server i in point-value form. That is, instead of sending each server shares of the coefficients of h, the client sends each server shares of evaluations of h. In particular, the client evaluates $[h]_i$ at all of the points $t \in \{1, \ldots, 2M-1\}$ and sends the evaluations $[h]_i(1), [h]_i(2), \ldots, [h]_i(2M-1)$ to the server. Now, each server i already has the evaluations of $[h]_i$ at all of the points it needs to complete Step 2 of the SNIP verification. To complete Step 3, each server must interpolate $[h]_i$ and evaluate $[h]_i$ at the point r. We accomplish this using the same fast interpolation-and-evaluation trick described above for $[f]_i$ and $[g]_i$

Circuit optimization In many cases, the servers hold multiple verification circuits $\mathsf{Valid}_1, \ldots, \mathsf{Valid}_N$ and want to check whether the client's submission passes all N checks. To do so, we have the Valid circuits return zero (instead of one) on success. If W_j is the value on the last output wire of the circuit Valid_j, we have the servers choose random values $(r_1, \ldots, r_N) \in \mathbb{F}^N$ and publish the sum $\sum_j r_j W_j$ in the last step of the protocol. If any $W_j \neq 0$, then this sum will be non-zero with high probability and the servers will reject the client's submission.

Opaque: An Oblivious and Encrypted Distributed Analytics Platform

Wenting Zheng, Ankur Dave, Jethro G. Beekman,
Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica
UC Berkeley

Abstract

Many systems run rich analytics on sensitive data in the cloud, but are prone to data breaches. Hardware enclaves promise data confidentiality and secure execution of arbitrary computation, yet still suffer from *access pattern leakage*. We propose Opaque, a distributed data analytics platform supporting a wide range of queries while providing strong security guarantees. Opaque introduces new distributed oblivious relational operators that hide access patterns, and new query planning techniques to optimize these new operators. Opaque is implemented on Spark SQL with few changes to the underlying system. Opaque provides data encryption, authentication and computation verification with a performance ranging from 52% faster to 3.3x slower as compared to vanilla Spark SQL; obliviousness comes with a 1.6–46x overhead. Opaque provides an improvement of *three orders of magnitude* over state-of-the-art oblivious protocols, and our query optimization techniques improve performance by 2–5x.

1 Introduction

Cloud-based big data platforms collect and analyze vast amounts of sensitive data such as user information (emails, social interactions, shopping history), medical data, and financial data. These systems extract value out of this data through advanced SQL [4], machine learning [25, 15], or graph analytics [14] queries. However, these information-rich systems are also valuable targets for attacks [16, 32].

Ideally, we want to both protect data confidentiality and maintain its value by supporting the existing rich stack of analytics tools. Recent innovation in trusted hardware enclaves (such as Intel SGX [24] and AMD Memory Encryption [19]) promise support for arbitrary computation [6, 34] at processor speeds while protecting the data.

Unfortunately, enclaves still suffer from an important attack vector: *access pattern leakage* [41, 28]. Such leakage occurs at the *memory level* and the *network level*. Memory-level access pattern leakage happens when a compromised OS is able to infer information about the encrypted data by monitoring an application's page accesses. Previous work [41] has shown that an attacker can extract hundreds of kilobytes of data from confidential documents in a spellcheck application, as well as discernible outlines of jpeg images from an image processing application running inside the enclave. Network-level access pattern leakage occurs in the distributed setting because tasks (e.g., sorting or hash-partitioning) can produce network traffic that reveals information about the encrypted data (e.g., key skew), even if the messages sent over the network are encrypted. For example, Ohrimenko et al [28] showed that an attacker who observes the metadata of network messages, such as source and destination (but not their content), in a MapReduce computation can identify the age group, marital status, and place of birth for some rows in a census database. Therefore, to truly secure the data, the computation should be *oblivious*: i.e., it should not leak any access patterns.

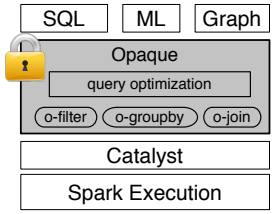


Figure 1: Opaque efficiently executes a wide range of distributed data analytics tasks by introducing SGX-enabled oblivious relational operators that mask data access patterns and new query optimization techniques to reduce performance overhead.

In this paper, we introduce Opaque[1], an oblivious distributed data analytics platform. Utilizing Intel SGX hardware enclaves, Opaque provides strong security guarantees including computation integrity and obliviousness.

One key question when implementing the oblivious functionality is: at what layer in the software stack should we implement it? Implementing at the application layer will likely result in application-specific solutions that are not widely applicable. Implementing at the execution layer, while very general, provides us with little semantics about an application beyond the execution graph and significantly reduces our ability to optimize the implementation. Thus, neither of these two natural approaches appears satisfactory.

Fortunately, recent developments and trends in big data processing frameworks provide us with a compelling opportunity: the *query optimization layer*. Previous work has shown that the relational model can express a wide

[1]The name "Opaque" stands for Oblivious Platform for Analytic QUEries, as well as opacity, hiding sensitive information.

variety of big data workloads, including complex graph analytics [14] and machine learning [17]. We chose to implement Opaque at this layer. While the techniques we present in this paper are general, we instantiate them using Apache Spark [4] by layering Opaque on top of Catalyst, the Spark SQL query optimizer (see Fig. 1). Our design requires no changes to Spark's libraries and requires minimal extensions to Catalyst.

The main challenge we faced in designing Opaque is the question of how to *efficiently* provide access pattern protection. It has long been known in the literature that such protection brings high overheads. For example, the state-of-the-art framework for oblivious computation, ObliVM [22], has an overhead of 9.3×10^6x and is not designed for distributed workloads. Even GraphSC [27], a special-purpose platform for oblivious parallel graph computation, reports a 10^5x slowdown.

To address this challenge, we propose a two-part solution. First, we introduce a set of new distributed relational operators that protect against both memory and network access pattern leakage at the same time. These include operators for joins and group-by aggregates. The contribution of these relational operators is to achieve obliviousness in a *distributed and parallel* setting. One recurring challenge here is to handle boundary conditions (when a value that repeats in rows spans multiple machines) in a way that is efficient and does not leak access patterns. These operators also come with computation integrity guarantees, called *self-verifying computation*, preventing an attacker from affecting the computation result.

Second, we provide novel query planning techniques, both rule-based and cost-based, to further improve the performance of oblivious computation.

- *Rule-based optimization.* Oblivious SQL operators in Opaque consist of fine-grained oblivious computation blocks called Opaque operators. We observe that by taking a global view across these Opaque operators and applying Opaque-specific rules, some operators can be combined or removed while preserving security.
- *Cost-based optimization.* We develop a cost model for oblivious operators that lets us evaluate the cost of a physical plan. This model introduces security as a new dimension to query optimization. We show that it is possible to achieve significant performance gains by using join reordering to minimize the number of oblivious operators. One key aspect used by our cost model is that *not all* tables in a database are sensitive: some contain public information. Hence, we can query such tables using non-oblivious operators to improve performance. Opaque allows database administrators to specify which tables are sensitive. However, sensitive tables can be related with seemingly insensitive tables. To protect the sensitive tables in this case, Opaque leverages a technique in the database literature called inference detection [18, 9] to propagate sensitivity through tables based on their schema information. Additionally, Opaque propagates operator sensitivity as well for all operators that touch sensitive tables.

We implemented Opaque using Intel SGX on top of Spark SQL with minimal modifications to Spark SQL. Opaque can be run in three modes: in *encryption mode*, Opaque provides data encryption and authentication as well as guarantees the correct execution of the computation; in *oblivious mode*, Opaque additionally provides oblivious execution that protects against access pattern leakage; in *oblivious pad mode*, Opaque improves on the oblivious mode by preventing size leakage.

We evaluate Opaque on three types of workloads: SQL, machine learning, and graph analytics. To evaluate SQL, we utilize the Big Data Benchmark [1]. We also evaluated Opaque on least squares regression and PageRank. In a 5-node cluster of SGX machines, encryption mode's performance is competitive with the baseline (unencrypted and non-oblivious): it ranges from being 52% faster to 3.3x slower. The performance gains are due to C++ execution in the enclave versus the JVM in untrusted mode (for vanilla Spark SQL). Oblivious mode slows down the baseline by 1.6–46x. Much of the oblivious costs are due to the fact that Intel SGX is not set up for big data analytics processing; future architectures [8, 21, 35] providing larger and oblivious enclave memory will reduce this cost significantly. We compare Opaque with GraphSC [27], a state-of-the-art oblivious graph processing system, by evaluating both systems on PageRank. Opaque is able to achieve three orders of magnitude (2300x) of performance gain, while also providing general SQL functionality. Finally, while obliviousness is fundamentally costly, we show that our new query optimization techniques achieve a performance gain of 2–5x.

2 Background

Opaque combines advances in secure enclaves with the Spark SQL distributed relational dataflow system. Here we briefly describe these two technologies, as well as exemplify an access pattern leakage attack.

2.1 Hardware Enclaves

Secure enclaves are a recent advance in computer processor technology providing three main security properties: fully isolated execution, sealing, and remote attestation. The exact implementation details of these properties vary by platform (e.g. Intel SGX [24] or AMD Memory Encryption [19]), but the general concepts are the same. Our design builds on the general notion of an enclave, which has several properties. First, *isolated execution* of an enclave process restricts access to a subset of memory such that only that particular enclave can access it. No

other process on the same processor, not even the OS, hypervisor, or system management module, can access that memory. Second, *sealing* enables encrypting and authenticating the enclave's data such that no process other than the exact same enclave can decrypt or modify it (undetectably). This enables other parties, such as the operating system, to store information on behalf of the enclave. Third, *remote attestation* is the ability to prove that the desired code is indeed running securely and unmodified within the enclave of a particular device.

2.2 Access pattern leakage attacks

To understand access pattern leakage concretely, consider an example query in the medical setting:

```
SELECT COUNT(*) FROM patient WHERE age > 30
GROUP BY disease
```

The "group by" operation commonly uses hash bucketing: each machine iterates through its records and assigns each record to a bucket. The records are then shuffled over the network so that records within the same bucket are sent to the same machine. For simplicity, assume each bucket is assigned to a separate machine. By watching network packets, the attacker sees the number of items sent to each machine. Combined with public knowledge about disease likelihood, the attacker infers each bucket's disease type.

Moreover, the attacker can learn the disease type for a *specific database record*, as follows. By observing page access patterns, the attacker can track a specific record's bucket assignment. If the bucket's disease type is known, then the record's disease type is also known. A combination of page-based access patterns and network-level access patterns thus gives attackers a powerful tool to gain information about encrypted data.

2.3 Spark background

We implemented Opaque on top of Spark SQL [42, 4], a popular cluster computing framework, and we use Spark terminology in our design for concreteness. We emphasize that the design of Opaque is not tied to Spark or Spark SQL: the oblivious operators and query planning techniques are applicable to other relational frameworks.

The design of Spark SQL [42, 4] is built around two components: master and workers. The user interacts with the master which is often running with the workers in the cloud. When a user issues a query to Spark SQL, the command is sent to the master which constructs and optimizes a physical query plan in the form of a DAG (directed acyclic graph) whose nodes are tasks and whose edges indicate data flow. The conversion of the SQL query into a physical query plan is mediated by the Catalyst query optimizer.

3 Overview

3.1 Threat model and assumptions

We assume a powerful adversary who controls the cloud provider's software stack. As a result, the adversary can observe and modify the network traffic between different nodes in the cloud as well as between the cloud and the client. The attacker may gain root access to the operating system, modify data or communications that are not inside a secure enclave, and observe the content and order of memory accesses by an enclave to untrusted memory (i.e., memory that is not part of a secure enclave). In particular, the adversary may perform a *rollback attack*, in which it restores sealed data to a previous state.

We assume the adversary cannot compromise the trusted hardware, relevant enclave keys, or client software. In particular, the attacker cannot issue queries or change server-side data through the client. Denial-of-service attacks are out of scope for this paper. A cloud provider may destroy all customer data or deny or delay access to the service, but this would not be in the provider's interest. Customers also have the option to choose a different provider if necessary. Side-channel attacks based on power analysis or timing attacks (including those that measure the time spent in the enclave or the time when queries arrive) are also out of scope.

We assume that accesses to the source code of Opaque that runs in the enclave are oblivious. This can be achieved either by making accesses oblivious using tools such as GhostRider [21], or by using an enclave architecture that provides a pool of oblivious memory [8, 21, 35]; the latter need only provide a small amount of memory because the relevant Opaque source code is $\approx$ 1.4MB.

3.2 Opaque's architecture

Figure 2 shows Opaque's architecture. Opaque does not change the layout of Spark and Spark SQL, except for one aspect. Opaque moves the query planner to the client side because a malicious cloud controlling the query planner can result in incorrect job execution. However, we keep the scheduler on the server side, where it runs in the untrusted domain. We augment Opaque with a computation verification mechanism (§4.2) to prevent an attacker from corrupting the computation results.

The Catalyst planner resides in the job driver and is extended with Opaque optimization rules. Given a job, the job driver outputs a task DAG and a unique job identifier JID for this job. For example, the query from §2.2 translates to the DAG shown in Fig. 3. The job driver annotates each edge with an ID, e.g., E1, and each node with a task ID, e.g., task 4. The input data is split in partitions, each having its own identifier.

Oblivious memory parameter. As discussed, the current Intel SGX architecture leaks memory access patterns

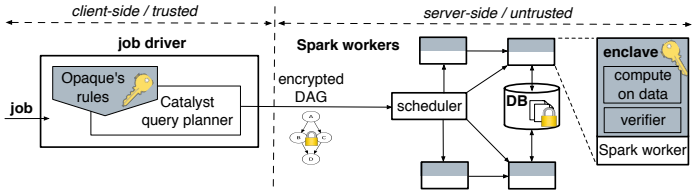


Figure 2: Opaque's architecture overview.

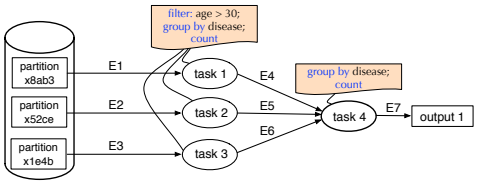


Figure 3: Example task DAG.

both when accessing the enclave's memory (EPC) and the rest of main memory. Nevertheless, recent work, such as Sanctum [8], GhostRider [21], and T-SGX [35], proposes enclave designs that protect against access patterns to the EPC. Hence, such systems yield a pool of oblivious memory, which can be used as a cache to speed up oblivious computations. Since the size of the oblivious pool depends on the architecture used, we parameterize Opaque with a variable specifying the size of the oblivious memory. This parameter can range from as small as the registers (plus Opaque's enclave code size) to as large as the entire EPC [8, 35] or main memory. Bigger oblivious memory allows faster oblivious execution in Opaque. In all cases, Opaque provides oblivious accesses to the non-oblivious part of the EPC, to the rest of RAM, and over the network.

3.3 Security guarantees

Encryption mode. In encryption mode, Opaque provides data encryption and authentication guarantees. Opaque's self-verifying integrity protocol (§4.2) guarantees that, if the client verifies the received result of the computation successfully, then the result is correct, i.e., not affected by a malicious attacker. The proof of security for the self-verifying integrity protocol is rather straightforward, and similar to the proof for VC3 [34].

Oblivious modes. In the two oblivious modes, Opaque provides the strong guarantee of oblivious execution with respect to memory, disk, and network accesses for every *sensitive SQL operator*. As explained in §6.3, these are operators taking as input at least one sensitive table or intermediate results from a set of operators involving at least one sensitive table. Opaque does not hide the computation/queries run at the server or data sizes, but it protects the data content. In oblivious mode, the attacker learns the size of each input and output to a SQL operator and the query plan chosen by Catalyst, which might leak some statistical information. The oblivious pad mode, explained in §5.3, hides even this information by pushing up all filters and padding the final output to a public upper bound, in exchange for more performance overhead. We formalize and prove our obliviousness guarantees in the extended version of this paper, and present only the statement of the guarantees here.

Consider oblivious mode. The standard way to formalize that a system hides access patterns is to exhibit a simulator that takes as input a query plan and data sizes but *not the data content*, yet is able to produce the same trace of memory and network accesses as the system. Intuitively, since the simulator did not take the data as input, it means that the accesses of the system do not depend on the data content. Whatever the simulator takes as input is an upper bound on what the system leaks.

To specify the leakage of Opaque, consider the following (informal) notation. Let $\mathcal{D}$ be a dataset and Q a query. Let $\mathsf{Size}(\mathcal{D})$ be the sizing information of $\mathcal{D}$, which includes the size of each table, row, column, attribute, the number of rows, the number of columns, but does not include the value of each attribute. Let S be the schema information, which includes table and column names in $\mathcal{D}$, as well as which tables are sensitive. Opaque can easily hide table and column names via encryption. The sensitive tables include those marked by the administrator, as well as those marked by Opaque after sensitivity propagation (§6.3). Let $\mathsf{IOSize}(\mathcal{D},Q)$ be the input/output size of each SQL operator in Q when run on $\mathcal{D}$. We define $P = \mathsf{OpaquePlan}(\mathcal{D},Q)$ to be the physical plan generated by Opaque. We define Trace to be the trace of memory accesses and network traffic patterns (the source, destination, execution stage, and size of each message) for sensitive operators.

Theorem 1. *For all $\mathcal{D}$, S, where $\mathcal{D}$ is a dataset and S is its schema, and for each query Q, there exists a polynomial-time simulator* Sim *such that, for* $P = \mathsf{OpaquePlan}(\mathcal{D},Q)$,

$$\mathsf{Sim}(\mathsf{Size}(\mathcal{D}),S,\mathsf{IOSize}(\mathcal{D},Q),P) = \mathsf{Trace}(\mathcal{D},P).$$

The existence of Sim demonstrates that access patterns of the execution are oblivious, and that the attacker does not learn the data content $\mathcal{D}$ beyond sizing information and the query plan. The fact that the planner chose a certain query plan over other possible plans for the same query might leak some information about the statistics on the data maintained by the planner. Nevertheless, the planner maintains only a small amount of such statistics that contain much less information than the actual data content. Further, the attacker does not see these statistics directly and does not have the power to change data or queries and observe changes to the query plan.

Oblivious pad mode's security guarantees are similar to the above, except that the simulator no longer takes as input $\mathsf{IOSize}(\mathcal{D},Q)$, but instead only a public upper bound on the size of a query's final output.

Note that Opaque protects most constants in a query using semantic security: for example it hides the constant in "age ≥ 30", but not in "LIMIT 30".

Coupling oblivious accesses with the fact that the content of every write to memory and every network message is freshly encrypted with semantic security enables Opaque to provide a strong degree of data confidentiality. In particular, Opaque protects against the memory and network access patterns attacks presented in [41] and [28].

4 Opaque's encryption mode

In this section, we describe Opaque's encryption mode, which provides data encryption, authentication and computation integrity.

4.1 Data encryption and authentication

Similar to previous designs [6, 34], Opaque uses remote attestation to ensure that the correct code has been loaded into enclaves. A secure communication channel is then established and used to agree upon a shared secret key k between the client and the enclaves.

All data in an enclave is automatically encrypted by the enclave hardware using the processor key of that enclave. Before communicating with another enclave, an enclave always encrypts its data with AuthEnc using the shared secret key k. AuthEnc encrypts data with AES in GCM mode, a high-speed mode that provides authenticated encryption. In addition to encryption, this mode also produces a 128-bit MAC to be used for checking integrity.

4.2 Self-verifying computation

Ensuring computation integrity is necessary because a malicious OS could drop messages, alter data or computation. We call our integrity checking strategy *self-verifying computation* because the computation verifies itself as it proceeds. The mere fact that the computation finished without aborting means that it was not tampered with.

Let us first discuss how to check that the input data was not corrupted. As in VC3 [34], the identifier of a partition of input data is its MAC. The MAC acts as a *self-certifying* identifier because an attacker cannot produce a different partition content for a given ID. Finally, the job driver computes $C \leftarrow \text{AuthEnc}_k(\text{JID}, \text{DAG}, P_1, \ldots, P_p)$, where $P_1, \ldots, P_p$ indicates the identifiers of the partitions to be taken as input. Every worker node receives C. Opaque's verifier running in the enclave decrypts and checks the authenticity of the DAG in C.

Then, to verify the integrity of the computation, each task needs to check that the computation up to it has proceeded correctly. First, if $E_1, \ldots, E_t$ are edges incoming into task T in the DAG, the verifier checks that it has received authentic input on each edge from the correct previous task and that it has received input for *all* edges. To ensure this invariant, each node producing an output o for an edge E encrypts this output using $\text{AuthEnc}_k(\text{JID}, E, o)$. The receiving node can check the authenticity of this data and that it has received data for *every* edge in the DAG. Second, the node will run the correct task T because the enclave code was set up using remote attestation and task T is integrity-verified in the DAG. Finally, each job ends with the job driver receiving the final result and checking its MAC. The last MAC serves as a proof of correct completion of this task.

This protocol improves over VC3 [34], which requires an extra stage where all workers send their inputs and outputs to a master which checks that they all received complete and correct inputs. Opaque avoids the cost of this extra stage and performs the verification during the computation, resulting in negligible cost.

Rollback attacks. Spark's RDDs combined with our verification method implicitly defend against rollback attacks, because the input to the workers is matched against the expected MACs from the client and afterwards, the computation proceeds deterministically. The computation result is the same even with rollbacks.

4.3 Fault tolerance

In Spark, if the scheduler notices that some machine is slow or unresponsive, it reassigns that task to another machine. Opaque's architecture facilitates this process because the encrypted DAG is *independent* from the workers' physical machines. As a result, the scheduler can live entirely in the untrusted domain, and does not affect Opaque's security if compromised.

5 Oblivious execution

In this section, we describe Opaque's oblivious execution design. We first present two oblivious building blocks, followed by Opaque's oblivious SQL operator designs.

5.1 Oblivious building blocks

Oblivious sorting is central to the design of oblivious SQL operators. Opaque adapts existing oblivious sorting algorithms for both local and distributed sorting, which we now explain.

5.1.1 Intra-machine oblivious sorting

Sorting networks [7] are abstract networks that consist of a set of *comparators* that compare and swap two elements. Elements travel over wires from the input to comparators, where they are sorted and output again over wires. Sorting networks are able to sort any sequence of elements using a fixed set of comparisons.

Denote by OM, the oblivious memory available for query processing, as discussed in §3.2. In the worst case, this is only a part of the registers. If the total size of the data to be sorted on a single machine fits inside the OM, then it is possible to load everything into the OM,

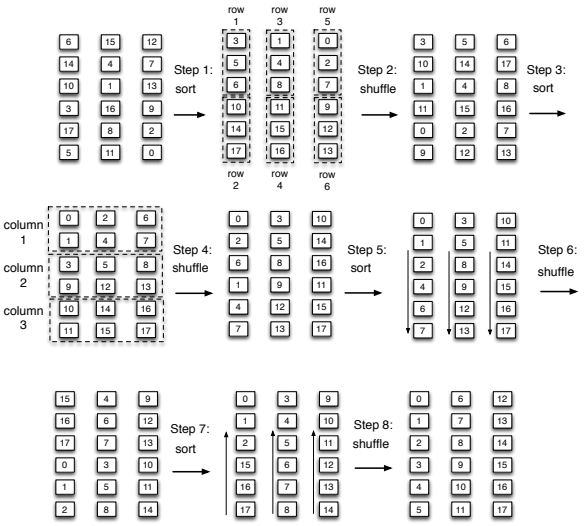


Figure 4: Column sort, used in the distributed setting. Each column represents a single partition, and we assume that each machine only has one partition. The algorithm has eight steps. Steps 1, 3, 5, 7 are sorts, and the rest are shuffle operations.

sort using quicksort, then re-encrypt and write out the result. If the data cannot fit inside the OM, Opaque will first partition the data into blocks. Each block is moved into the OM and sorted using quicksort. We then run a sorting network called *bitonic sort* over the blocks, treating each one as an abstract element in the network. Each comparator operation loads two blocks into the enclave, decrypts, merges, and re-encrypts the blocks. The merge operation only requires a single scan over the blocks.

5.1.2 Inter-machine oblivious sorting

A natural way to adapt the bitonic sorting network in the distributed setting is to treat each machine as an abstract element in the sorting network. We can sort within each machine separately, then run the bitonic sorting network over the machines. However, each level of comparators now corresponds to a network shuffling of data. Given n machines, the sorting network will incur $O(\log^2 n)$ number of shuffles, which is high.

Instead, Opaque uses column sort [20], which sorts the data using a *fixed* number of shuffles (5 in our experiments) by exploiting the fact that a single machine can hold many items. Column sort works as follows: given a sequence of B input items, we split these items into s partitions, where each partition has exactly r items (with padding if necessary). Without loss of generality, we assume that each machine handles one partition. We treat each partition as a column in column sort. The sorting algorithm has 8 steps: the odd-numbered steps are per-column sorts (implemented as intra-machine oblivious sorting), and the even-numbered steps shuffle the data deterministically. Figure 4 gives a visual example of how column sort works. The sorting algorithm has the restriction that $r \geq 2(s-1)^2$, which applies well to our setting because there are many records in a single partition/column.

An important property of column sort is that, as an oblivious operator, it preserves the *balance* of the partitions. This means that after a sort, a partition will have exactly the same number of items as before. Partition balance is required to avoid leaking any information regarding the underlying data's distribution. However, balanced partitioning is incompatible with co-locating all records of a given group. Instead, records with identical grouping attributes may be split across partitions. Operators that consume the output of column sort must therefore be able to transfer information between adjacent partitions *obliviously* and *efficiently*. We address this challenge in our descriptions of the oblivious operators.

5.2 Oblivious operators

In this section, we show how to use the oblivious building blocks to construct oblivious relational algebra operators. The three operators we present are filter, group-by, and join. Opaque uses an existing oblivious filter operator [3], but provides new algorithms for the join and group-by operators, required by the distributed and parallel setting.

In what follows, we focus only on the salient parts of these algorithms. We do not delve into how to make simple structures oblivious like conditionals or increments, which is already known (e.g., [21]).

5.2.1 Oblivious filter

An oblivious filter ensures that the attacker cannot track which encrypted input rows pass the filter. A naïve filter that streams data through the enclave to get rid of unwanted rows will leak which rows have been filtered out because the attacker can keep track of which input resulted in an output. Instead, the filter operator [3] used in Opaque first scans and marks each row with a "0" (record should be kept) or a "1" (record should be filtered), then obliviously sorts all rows with "0" before "1", and lastly, removes the "1" rows.

5.2.2 Oblivious Aggregate

Aggregation queries group items with equal *grouping attributes* and then aggregate them using an aggregation function. For example, for the query in §2.2, the grouping attribute is `disease` and the aggregation function is `count`.

A naïve aggregation implementation leaks information about group sizes (some groups may contain more records than others), as well as the actual mapping from a record to a group. For example, a reduce operation that sends all rows in the same group to a single machine reveals which and how many rows are in the group. Prior work [28] showed that an attacker can identify age group or place of birth from such protocols.

Opaque's oblivious aggregation starts with an oblivious sort on the grouping attributes. Once the sort is complete,

all records that have the same grouping attributes are located next to each other. A single scan might seem sufficient to aggregate and output a value for each group, but this is incorrect. First, the number of groups per machine can leak the number of values in each group. A further challenge (mentioned in §5.1.2) is that a set of rows with the same grouping attributes might span multiple machines, leaking such information. We need to devise a parallel solution because a sequential scan is too slow.

We solve the above problems by designing a distributed group-by operator that reveals neither row-to-group mapping nor the size of each group. The logical unit for this algorithm is a partition, which is assumed to fit on one machine. The intuition for this algorithm is that we want to *simulate* a global sequential scan using per-partition parallel scans. If all records in a group are in one partition, the group will be aggregated immediately. Once the last record in that group has been consumed in the scan, the aggregation result is complete. If records in a group are split across partitions, we want to pass information across partitions efficiently and obliviously so that later partitions have the information they need to finish the aggregation.

High-cardinality aggregation. This aggregation algorithm should be run when the number of groups is large.

Stage 1 [sort]: Obliviously sort all records based on the grouping attributes.

Stages 2–4 are the *boundary processing* stages. These stages solve the problem of a single group being split across multiple machines after column sort. Figure 5 illustrates an example.

Stage 2 [per-partition scan 1]: Each worker scans its partition once to gather some statistics, which include the partition's first and last rows, as well as partial aggregates of the last group in this partition. In Figure 5, each column represents one partition. Each worker calculates statistics including R_i, the partial aggregate. In partition 0, $R_0 = (C, 2)$ is the partial aggregate that corresponds to the last row in that partition, C.

Stage 3 [boundary processing]: All of the statistics from stage 2 are collected into a single partition. The worker assigned this partition will scan all of the statistics and compute one global partial aggregate (GPA) per partition. Each partition's GPA should be given to the next partition.

Figure 5's stage 3 shows an example of how the GPA is computed. The first partition always receives a dummy GPA since it is not preceded by any other partition. Partition $P1$ receives $(C, 2)$ from $P0$. With this information, $P1$ can correctly compute the aggregation result for group C, even though the records are split across $P0$ and $P1$.

Stage 4 [per-partition scan 2]: Each partition receives a GPA, which will be used to produce the final aggregation results. Figure 5's stage 4 shows that $P1$ can aggregate groups C, D and E using $R_1^{'}$. Note that one record needs

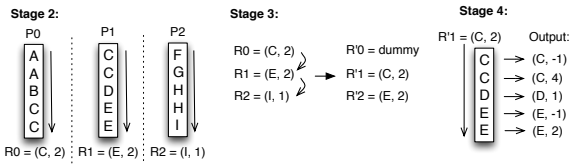


Figure 5: Stages 2 - 4 of oblivious aggregation

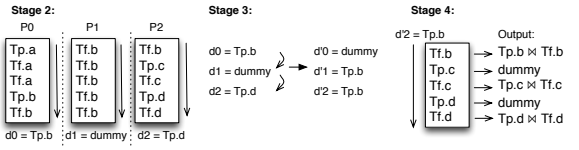


Figure 6: Stages 2–4 of oblivious join.

to be output for every input record, and output rows are marked as dummy if necessary (e.g., returning -1 for the count result).

Stage 5 [sort and filter]: Obliviously sort the dummy records after the real records, and filter out the dummies.

Low-cardinality group-by. If the number of groups is small (e.g., age groups, states), Opaque provides an alternative algorithm that avoids the second oblivious sort, which we describe in the extended version of this paper.

5.2.3 Oblivious sort-merge join

Regular joins leak information about how many and which records are joined together on the same join attributes. For example, a regular primary-foreign key join may sort the two tables separately, maintain a pointer to each table, and merge the two tables together while advancing the pointers. The pointer locations reveal information about how many rows have the same join attributes and which rows are joined together.

We developed an oblivious equi-join algorithm based on the sort-merge join algorithm. While our algorithm presented below focuses on primary-foreign key join, we can also generalize the algorithm to inner equi-join, which we describe in our extended paper. Let T_p be the primary key table, and T_f be the foreign key table.

Stage 1 [union and sort]: We union T_p with T_f, then obliviously sort them together based on the join attributes. We break ties by ordering T_p records before T_f records.

As with oblivious aggregation, stages 2–4 are used to handle the case of a join group (e.g., a set of rows from T_p and T_f that are joined together) that is split across multiple machines. We use Figure 6 to illustrate these three stages.

Stage 2 [per-partition scan 1]: Each partition is scanned once and the last row from T_p in that partition, or a dummy (if there is *no* record from T_p on that machine) is returned. We call this the boundary record.

Figure 6 explains stage 2 with an example, where Tp.x indicates a record from the primary key table with join

attribute x, and Tf.x indicates a record from the foreign key table with join attribute x. In partition $P0$, Tp.b is the last record of T_p in that partition, so the boundary record is set to Tp.b. $P1$ does not contain any row from T_p, so its boundary record is set to a dummy value.

Stage 3 [boundary processing]: In stage 3, we want to generate primary key table records to give back to each data partition so that all of the foreign key table records in each partition (even if the information spans across multiple machines) can be joined with the corresponding primary key record. We do so by first collecting all of the boundary records to one partition. This list is scanned once, and we output a new boundary record for every partition. Each output is set to the value of the most recently encountered *non-dummy* boundary.

For example, Fig. 6's stage 3 shows that three boundary records are collected. Partition 0 will always get a dummy record. Record Tp.b is passed from partition 0 to partitions 1 and 2 because d_1 is a dummy. This ensures that any record from T_f with join attribute b (e.g., the first record of partition 2) will be joined correctly.

Stage 4 [per-partition scan 2]: Stage 4 is similar to a normal sort-merge join, where the worker linearly scans the tables and joins primary key records with the corresponding foreign key records. There are some variations to preserve obliviousness. First, the initial record in the primary key table should come from the boundary record received in stage 3 (except for the first partition). Second, during the single scan, we need to make sure that one record is output for every input record, outputting dummy records as necessary.

Figure 6's stage 4 shows how the algorithm works on partition 2. The boundary record's value is Tp.b, which is successfully joined with the first row of partition 2. Since P_2's second row is a new record from T_p, we change the boundary record to Tp.c, and a dummy is output.

Stage 5 [sort and filter]: Oblivious sort to filter out the dummies.

5.3 Oblivious pad mode

Oblivious execution provides strong security guarantees and prevents access pattern leakage. However, it does not hide the output *size* of each relational operator. This means that in a query with multiple relational operators, the size of each intermediate result is leaked. To solve this problem, Opaque provides a stronger variant of oblivious execution: oblivious with padding.

The idea is to never reduce the output size of a relational operator until the end of the query. This can be easily achieved by using "filter push up." For example, a query that has a join followed by an aggregation will skip stage 5 of the join. After the aggregation, all dummies will be filtered out in a single sort with filter. We also require the user to provide an upper bound on the final result size, and Opaque will pad the final result to this size. In this case, the query plan also no longer depends on data statistics, as we discuss in §6.4.

Note that this mode is more inefficient because Opaque cannot take advantage of selectivity (e.g., of filters), and we provide an evaluation in our extended paper. Therefore, we recommend using padding on extremely sensitive datasets.

6 Query planning

Even with parallelizable oblivious algorithms, obliviousness is still expensive. We now describe Opaque's query planner, which reduces obliviousness overheads by introducing novel techniques that build on rule-based and cost-based optimization, as well as entity-relational modeling. We first formalize a cost model for our oblivious operators to allow a standard query planner to perform basic optimizations on oblivious plans. We then describe several new optimizations specific to Opaque, enabled by a decomposition of oblivious relational operators into lower-level Opaque operators. Finally, we describe a mixed sensitivity setting where a database administrator can designate tables as sensitive. Opaque applies a technique in databases known as second path analysis that uses foreign-key relationships in a data model to identify tables that are *not* sensitive, accounting for inference attacks. We also demonstrate that such sensitivity propagation occurs within a single query plan, allowing us to substantially speed up certain queries using join reordering.

6.1 Cost model

Cost estimation in Opaque differs from that of a traditional SQL database because sorting, the core database operation, is more costly in the oblivious setting than otherwise. Oblivious sorting has very different algorithmic behavior from conventional sorting algorithms because the sequence of comparisons can be constructed based only on the input *size* and not the input data. Therefore, our cost model must accurately model oblivious sorting, which is the dominant cost in our oblivious operators.

Similarly to a conventional sort, the cost of an oblivious sort depends on two factors: the number of input items and the padded record size. Even for datasets that fit in memory, cost modeling for an oblivious sort is similar to that of a traditional external sort because the latency penalty incurred by the enclave for accessing pages outside of the oblivious memory or EPC effectively adds a layer to the memory hierarchy. We therefore use a two-level sorting scheme for oblivious sort, described in §5.1.1, having a runtime complexity of $O(n \log^2 n)$.

We now formalize the cost of oblivious sort and use this to model oblivious join. The costs of other oblivious operators can be similarly modeled.

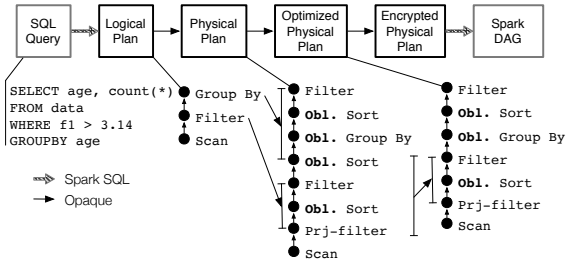


Figure 7: Catalyst oblivious query planning.

Let T be a relation, and r be a padded record. We denote $|T|$ to be the size of the relation T, and $|r|$ to be the size of a padded record. Let |OMem| be the size of the oblivious memory, and K a constant scale factor representing the cost of executing a compare-and-swap on two records. We denote n to be the number of records per block, and B to be the required number of blocks. We can estimate n, B, and the resulting sort cost $C_{\text{o-sort}}$ and join cost $C_{\text{o-join}}$ as follows:

$$n = \frac{|\text{OMem}|}{2\,|R|}, \quad B = |T|/n, \quad C_{\text{o-join}} \approx 2 \cdot C_{\text{o-sort}}$$

$$C_{\text{o-sort}}(|T|,|R|) = \begin{cases} K\,|T|\log|T| & \text{if } |T|\cdot|R| \le |\text{OMem}| \\ K\,[Bn\log n + nB\log B(1+\log B)/2] & \text{otherwise} \end{cases}$$

The number of records n per block follows from the fact that two blocks must fit in oblivious memory at a time for the merge step. The expression for the sort cost follows from the two-level sorting scheme. If the input fits inside the oblivious memory, we bypass the sorting network and instead use quicksort within this memory, so the estimated cost is simply the cost of quicksort. Otherwise, we sort each block individually using quicksort, run a sorting network on the set of blocks and merge blocks pairwise. The sorting network performs $B\log B(1+\log B)/4$ merges, each incurring a cost of $2n$ to merge two blocks. We experimentally verify this cost model in §8.4.

6.2 Oblivious query optimization

We now describe new optimization rules for a sequence of oblivious operators. Our rules operate on the lower-level operations within each oblivious operator, which we call Opaque operators.

6.2.1 Overview of the query planner

Before describing the Opaque operators, we provide an overview of the planning process, illustrated in Fig. 7. Opaque leverages the Catalyst query planner to transform a *SQL query* into an operator graph encoding the *logical plan*. Opaque interposes in the planning process to mark all logical operators that process sensitive data as oblivious. Catalyst can apply standard relational optimizations to the logical plan such as filter pushdown and join reordering.

Catalyst then generates a *physical plan* where each logical operator is mapped to one or more physical operators representing the choice of execution strategy. For example, a logical non-oblivious join operator could be converted to a physical hash join or a broadcast join based on the input cardinalities. Oblivious operators are transformed into physical Opaque operators at this stage, allowing us to express rules specific to combinations of oblivious operators. Similar to Catalyst, generating these physical operators allows Opaque to select from multiple implementations of the same logical operator based on table statistics. For example, if column cardinality is available, Opaque may use it to decide which oblivious aggregation algorithm to use. Catalyst then applies our Opaque rules to the physical plan.

The physical plan is then converted into an *encrypted representation* to hide information such as column names, constants, etc. Finally, Catalyst transforms the encrypted physical plan into a *Spark DAG* containing a graph of RDDs and executes it on the cluster.

6.2.2 Opaque operators

The following is a sampling of the physical Opaque operators generated during planning:

- `SORT`(C): obliviously sort on columns C
- `FILTER`: drop rows if predicate not satisfied
- `PROJECT-f`: similar to `FILTER`, but projects filtered out rows to 1, the rest to 0; preserves input size
- `HC-AGG`: stages 2–4 of the aggregation algorithm
- `SORT-MERGE-JOIN`: steps 2–4 of the sort-merge join algorithm

6.2.3 Query optimization

In this section, we give an example of an Opaque-specific rule:

$$\texttt{SORT}(C_2, \texttt{FILTER}(\texttt{SORT}(C_1, \texttt{PROJECT-f}(C_1)))) = \texttt{FILTER}(\texttt{SORT}(C_1, C_2, \texttt{PROJECT-f}(C_1)))$$

Let us take a look at how this rule would work with a specific query. We use the example query from §2.2, which translates to the following physical plan:

```
LC-AGG(disease,
  SORT(disease, FILTER(dummy,
    SORT(dummy_col, PROJECT-f(age, patient)))))
```

The filter will first do a projection based on the column age. To preserve obliviousness, the projected column is sorted and a real filter is applied. Since a sort-based aggregation comes after the filter, we need to do another sort on disease.

We make the observation that the second sort can be combined with the first sort into *one* oblivious sort on multiple columns. Since `PROJECT-f` always projects a column that is binary (i.e., the column contains only "0"s and "1"s), we can first sort on the binary column, then on

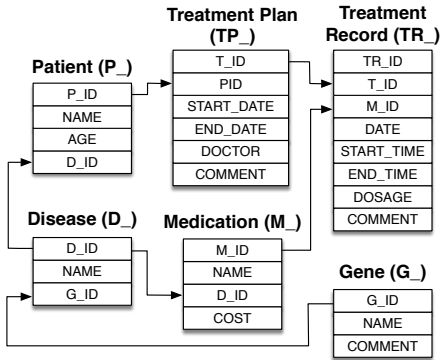


Figure 8: Example medical schema.

the second sort's columns (in this example, the disease column). Therefore, the previous plan becomes:

```
LC-AGG(disease, FILTER(dummy_col,
  SORT({dummy_col, disease},
    PROJECT-f(age, patient))))
```

This optimization is rule-based instead of cost-based. Furthermore, our rule is different from what a regular SQL optimizer applies because it pushes *up* the filter, while a SQL optimizer pushes *down* the filter. Filter push-down is unsafe because it does not provide obliviousness guarantees. Applying the filter before sorting will leak which records are filtered out.

6.3 Mixed sensitivity

Many applications operate on a database where not all of the tables are sensitive. For example, a hospital may treat patient information as sensitive while information about drugs, diseases, and various hospital services may be public knowledge (see Figure 8).

6.3.1 Sensitivity propagation

Propagation on tables. In a mixed sensitivity environment, tables that are not marked as sensitive could still be sensitive if they reveal information about other sensitive tables. Consider the example schema in Fig. 8. The Disease, Medication, and Gene tables are public datasets or have publicly known distributions in this example and therefore are not sensitive. Meanwhile the Patient table would likely be marked as sensitive. But what about Treatment Plan and Treatment Record? It turns out these tables are also sensitive because they *implicitly* embed patient information. Each treatment record belongs to a single patient, and each patient's plan may contain multiple treatment records. If an attacker has some prior knowledge, for example regarding what type of medication a patient uses, then observing only the Treatment Record table may allow the attacker to use an *inference attack* to gain further information about that patient such as their treatment frequency and other medication they may be taking.

To prevent such attacks, we use a technique from database literature called second path analysis [18]. The intuition for the inference attack is that information propagates along primary-foreign key relations: since each treatment record belongs to one treatment plan and one patient, the treatment record contains implicit information about patients. The disease table is connected to the patient table as well, except it has a primary key pointing *into* patient. This means that the disease table does not implicitly embed patient information.

Second path analysis accomplishes table sensitivity propagation by first directly marking user-specified tables as sensitive. After this is done, it recursively marks all tables that are reachable from every sensitive table via primary-foreign key relationships as sensitive as well. As in Fig. 8, such relationships are marked in an entity-relationship diagram using an arrow from the primary key table to the foreign key table.

This approach has been generalized to associations other than explicit foreign keys and implemented in automated tools [9]. We do not reimplement such analysis in Opaque, instead referring to the existing work.

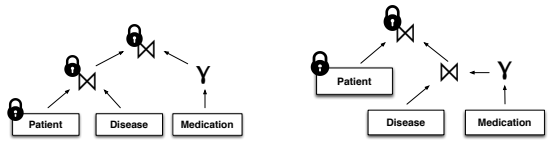
Propagation on operators. Another form of sensitivity propagation occurs when an operator (e.g., join) involves a sensitive and a non-sensitive table. In this case, we must run the entire operator obliviously. Additionally, for every leaf table that is marked sensitive in a query plan, sensitivity propagates on the path from the leaf to the root, and Opaque runs all the operators on this path obliviously.

6.3.2 Join reordering

Queries involving both sensitive and non-sensitive tables may contain a mix of oblivious and non-oblivious operators. Due to sensitivity propagation on operators, some logical plans may involve more oblivious operators than others. For example, a three-way join query where one table is sensitive may involve two oblivious joins if the sensitive table is joined first, or only one oblivious join if it is joined last (i.e., the non-sensitive tables are pushed down in the join order).

Join reordering in a traditional SQL optimizer centers on performing the most selective joins first, reducing the number of tuples that need to be processed. The statistics regarding selectivity can be collected by running oblivious Opaque queries. In Opaque, mixed sensitivity introduces another dimension to query optimization because of operator-level sensitivity propagation and the fact that oblivious operators are much more costly than their non-oblivious counterparts. Therefore, a join ordering that minimizes the number of oblivious operators may in some cases be more efficient than one that only optimizes based on selectivity.

Consider the following query to find the least costly medication for each patient, using the schema in Fig. 8:



(a) SQL join order (b) Opaque join order

Figure 9: Join reordering in mixed sensitivity mode.

```
SELECT    p_name, d_name, med_cost
FROM      patient, disease,
          (SELECT d_id, min(cost) AS med_cost
            FROM medication
            GROUP BY d_id) AS med
WHERE     disease.d_id = patient.d_id
          AND disease.d_id = med.d_id
```

We assume that the Patient table is the smallest, followed by Disease, then Medication ($|P| < |D| < |M|$), as might occur when considering only currently hospitalized patients and assuming there are multiple medications for each disease. The aggregation query reduces the cardinality of Medication to that of Disease and ensures a one-to-one relationship between the two tables.

Figure 9 shows two join orders for this query. A traditional SQL optimizer will execute the most selective join first, joining Patient with Disease, then with Medication. The optimal ordering for Opaque will instead delay joining Patient to reduce the number of oblivious joins. To see this, we now analyze the costs for both join orders.

Let C_{SQL} be the cost of this query using the SQL join order, C_{Opaque} the cost using the Opaque join order, and R the padded row size for all input tables. Note that the size of the Medication aggregate table is $|D|$.

$$C_{\text{SQL}} = 2C_{\text{o-join}}(|P| + |D|, R)$$
$$C_{\text{Opaque}} = C_{\text{join}}(2\,|D|, R) + C_{\text{o-join}}(|P| + |D|, R)$$

Assuming $C_{\text{join}} \ll C_{\text{o-join}}$,

$$\frac{C_{\text{SQL}}}{C_{\text{Opaque}}} \leq \frac{2C_{\text{o-join}}(|P| + |D|, R)}{C_{\text{o-join}}(|P| + |D|, R)} = 2$$

Thus, this query will see at most 2x speedup from join reordering. However, other queries can benefit still further from this optimization. Consider a three-way join of Patient, Disease, and Gene to extract the gene mutation affecting each patient. We assume Gene is a very large public dataset, so that $|P| < |D| < |G|$. Because Disease contains a foreign key into Gene, the three-way join occurs only on primary-foreign key constraints with no need for aggregation. As before, a traditional SQL optimizer would execute $(P \bowtie D) \bowtie G$ while Opaque will run $(G \bowtie D) \bowtie P$. The costs are as follows:

$$C_{\text{SQL}} = C_{\text{o-join}}(|P| + |D|, R) + C_{\text{o-join}}(|P| + |G|, R)$$
$$C_{\text{Opaque}} = C_{\text{join}}(|G| + |D|, R) + C_{\text{o-join}}(|D| + |P|, R)$$

Assuming $C_{\text{join}} \ll C_{\text{o-join}}$ and $|P| < |D| \ll |G|$,

$$\frac{C_{\text{SQL}}}{C_{\text{Opaque}}} = \frac{C_{\text{o-join}}(|P| + |G|, R)}{C_{\text{join}}(|G| + |D|, R)} \approx \frac{C_{\text{o-join}}(|G|, R)}{C_{\text{join}}(|G|, R)}$$

The maximum theoretical performance gain for this query therefore approaches the performance difference between the Opaque and non-oblivious join operators. We demonstrate this empirically in Fig. 12b.

Limitations. Note that sensitivity propagation optimizes efficiently when the large tables in a database are not sensitive. This makes intuitive sense because computation on larger tables contributes more to the query runtime. If the larger tables are sensitive, then join reordering cannot help because any join with these tables must always be made oblivious. Therefore, the underlying schema will have a large impact on the effectiveness of our cost-based query optimizations.

6.4 Query planning for oblivious pad mode

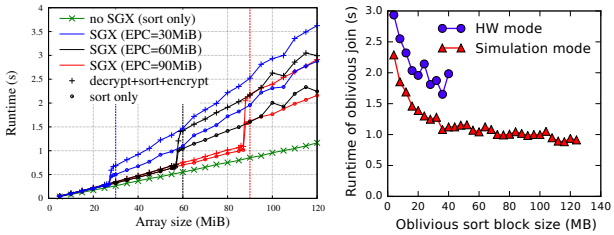
As discussed in §3.3, the fact that the planner chose a query plan over another plan leaks some information about the selectivity of some operators. For example, generalized inner joins' costs depend on join selectivity information. This is not a problem for primary-foreign key joins because these costs can be estimated using only the size of each table: the output size of such a join is always the size of the foreign key table.

Oblivious pad mode does not leak such statistics information. All filters are pushed up and combined together at the end of the query. The optimizer does not need to use selectivity information because the overall size will not be reduced until the very end. Thus, our query planning stage only needs to use publicly-known information such as the size of each table.

7 Implementation

Opaque is implemented on top of Spark SQL, a big data analytics framework. Our implementation consists of 7000 lines of C++ enclave code and 3600 lines of Scala.

We implemented the Opaque operators and query optimization rules from §6 by extending Catalyst using its developer APIs with minimal modifications to Spark. Our operators are written in Scala and execute in the untrusted domain, making trusted calls to the enclave when necessary through JNI. For example, the `SORT` operator performs inter-machine sorting using an RDD-based implementation of distributed column sort in the untrusted domain (§5.1.2). Within each partition, the `SORT` operator serializes the encrypted rows and passes them using JNI to the worker node's enclave, which then performs the local sort in the trusted domain (§5.1.1). Our implementation currently does not support arbitrary user-defined functions (UDFs) due to the difficulty in making them oblivious.



(a) Varying EPC size (b) Oblivious sort block size

Figure 10: Sort microbenchmarks. (a) Non-oblivious sort in SGX. Exceeding EPC size causes a dramatic slowdown. (b) Oblivious sort in SGX. Larger blocks improve performance until the EPC limit in HW mode, or indefinitely in simulation mode.

Opaque encrypts and integrity-protects data on a block-level basis using AES in GCM mode, which provides data confidentiality as well as integrity. We pad all rows within a table to the same upper bound before encrypting. This is essential for tables with variable-length attributes as it prevents an attacker from distinguishing between different rows as they move through the system.

8 Evaluation

In this section, we demonstrate that Opaque represents a significant performance improvement over the state of the art in oblivious computation, quantify its overhead compared to an insecure baseline, and measure the gains from our query planning techniques.

8.1 Experimental setup

Single-machine experiments were run using SGX hardware on a machine with Intel Xeon E3-1280 v5 (4 cores @ 3.70GHz, 8MiB cache) with 64GiB of RAM. This is the maximum number of cores available on processors supporting SGX at the time of writing.

Distributed experiments were run on a cluster of 5 SGX machines with Intel Xeon E3-1230 v5 (4 cores @ 3.40GHz, 8MiB cache) with 64GiB of RAM.

8.2 Impact of oblivious memory size

We begin by studying the impact of the secure enclave memory size and show that Opaque will benefit significantly from future enclave implementations with more memory. SGX maintains an encrypted cache of memory pages called the Enclave Page Cache, which is small compared to the size of main memory. Once a page is evicted from the EPC, it is decrypted if it was not entirely in CPU cache, re-encrypted under a different key, and stored in main memory. When an encrypted page in main memory is accessed, it needs to be decrypted again. This paging in and out of the EPC introduces a large overhead. Current implementations of SGX have a maximum effective EPC size of 93.5MiB, but this will be significantly increased in upcoming versions of SGX.

Sorting is the core operation in Opaque, so we studied how SGX affected its performance. In Fig. 10a, we benchmark non-oblivious sorting (introsort) in SGX by sorting arrays of 64-bit integers of various sizes using EPCs of various sizes. We also measure the overhead incurred by decrypting input data and encrypting output data before and after sorting using AES-GCM-128. We see that exceeding the EPC size even by just a little incurs a 50 ~ 60% overhead. When below the EPC limit, the overhead of encryption for I/O is just 7.46% on average. The overhead of the entire operation versus the insecure baseline is 31.7% on average.

Having a part of EPC that is oblivious radically improves performance. In §3.2, we discussed existing and upcoming designs for such an EPC. In Fig. 10b, we call this an oblivious block size, and we benchmarked the performance of oblivious sort with varying block sizes (§5.1.1). Within a block, regular quicksort can happen which speeds up performance. The case when only the registers are oblivious (namely an oblivious block of the same size as the available registers) did not fit in the graph: the overhead was 30x versus when the L3 cache (8MB) is oblivious. We see that in hardware mode, more oblivious memory improves performance until a sort block size of 40 MB, when the working set (two blocks for merging) exceeds the hardware EPC size, causing thrashing, as occurred in Fig. 10a near EPC limits. In simulation mode, no thrashing occurs. In sum, Opaque's performance will improve significantly when run with more oblivious memory as a cache.

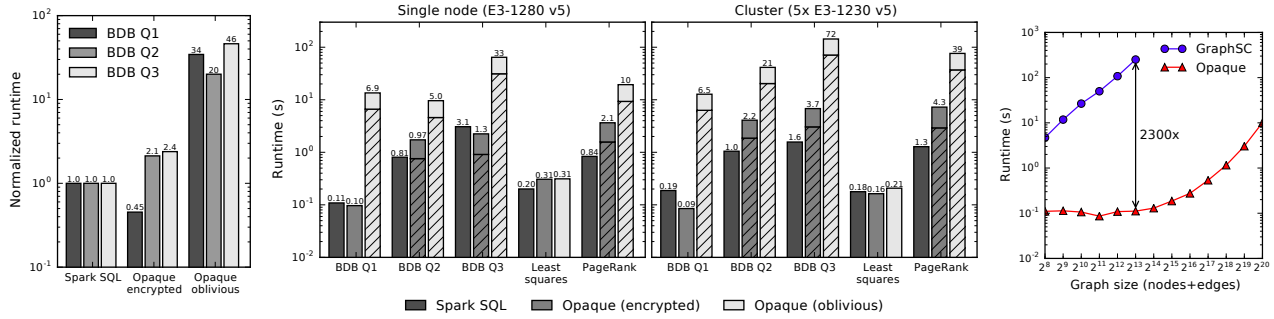
8.3 System comparisons

8.3.1 Comparison with Spark SQL

We evaluated Opaque against vanilla Spark SQL, which provides no security guarantees, on three different workloads: SQL, machine learning, and graph analytics.

For the SQL workload, we benchmarked both systems on three out of four queries of Big Data Benchmark [1], a popular benchmark for big data SQL engines. The fourth query is an external script query and is not supported by our system. The three queries cover filter, aggregation (high cardinality), and join. For the machine learning workload, we chose least squares regression on 2D data; this query uses projection and global aggregation. Finally, we chose to benchmark PageRank for the graph analytics workload; this query uses projection and aggregation.

We show our results in two graphs, Figure 11a and Figure 11b. Figure 11a shows the performance of each of Opaque's security modes on the Big Data Benchmark in the distributed setting. Higher security naturally adds more overhead. Encryption mode is competitive with Spark SQL (between 52% improvement and 2.4x slowdown). The performance gain comes from the fact that



(a) Security mode comparison (b) Comparison to Spark SQL (c) Comparison to GraphSC

Figure 11: (a) Encryption mode is competitive with Spark SQL. Obliviousness (including network and memory obliviousness) adds up to 46x overhead. (b) Comparison across a wide range of queries. Hatched areas represent time spent sorting. (c) Single iteration of PageRank for various graph sizes.

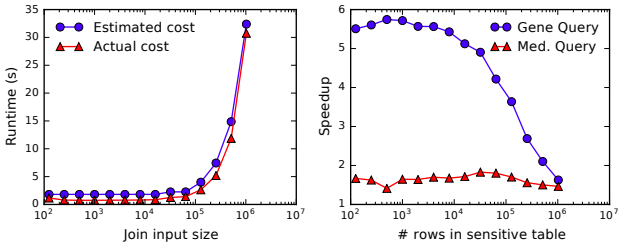
Opaque runs C++ in the enclave, while Spark SQL incurs overhead from the JVM. Opaque's oblivious mode adds 20–46x overhead.

Figure 11b shows Opaque's performance on five queries. Hatched areas show the time spent in oblivious sort, the dominant cost. The left side of Figure 11b shows Opaque running on a single machine using SGX hardware compared to Spark SQL, while the right side shows the distributed setting. In the single-machine setting, Opaque's encryption mode performance varies from 58% performance gain to 2.5x performance loss when compared with the Spark SQL baseline. The oblivious mode (both network and memory oblivious) slows down the baseline by 1.6–62x. The right side shows Opaque's performance on a distributed SGX cluster. Encryption mode's performance ranges from a 52% performance improvement to a 3.3x slowdown, while oblivious mode adds 1.2–46x overhead. In these experiments, Opaque was configured with oblivious memory being the L3 cache and not the bulk of EPC. As discussed in §8.2, more oblivious memory would give better performance, and such hardware proposals already exist (see §3.2).

8.3.2 Comparison with GraphSC

We use the same PageRank benchmark to compare with the existing state-of-the-art graph computation platform, GraphSC [27]. While Opaque is more general than graph computation, we compared Opaque with GraphSC instead of its more generic counterpart ObliVM [22], because ObliVM is about ten times slower than GraphSC.

We used data from GraphSC and ran the same experiment on both systems on our single node machine, with Opaque running in hardware mode with obliviousness. Figure 11c shows that Opaque is faster than GraphSC for all data sizes. For 8K graph size, Opaque is 2300x faster than GraphSC. This is consistent with the ObliVM and GraphSC papers: ObliVM reports a 9.3×10^6x slowdown, and GraphSC [27] a slowdown of 2×10^5x to 5×10^5x.



(a) Accuracy of obl. join cost model (b) Speedup from join reordering

Figure 12: Query planning benchmarks. (a) Our cost model closely approximates the empirical results for oblivious joins across a range of input sizes. (b) Join reordering provides up to 5x speedup for some queries.

Though GraphSC and Opaque share the high-level threat model of an untrusted service provider, they relax the threat model in different ways, explaining the performance gap. Opaque relies on trusted hardware, while GraphSC relies on two servers that must not collude and are semi-honest (do not cheat in the protocol) and so must use garbled circuits and secure two-party computation, which are much slower for generic computation than trusted hardware.

8.4 Query planning

We next evaluate the query planning techniques proposed in §6. First, we evaluate the cost model presented in §6.1 using a single-machine microbenchmark. We run an oblivious join and vary the input cardinality. We then fit the equation from §6.1 to the empirical results. Figure 12a shows that our theoretical cost model closely approximates the actual join costs.

Second, to evaluate the performance gain from join reordering, we run the two queries from §6.3.2. Figure 12b shows the speedup from reordering each query with varying sizes of the sensitive patient table. The medication query sees just under 2x performance gain because two equal-sized oblivious joins are replaced by

one oblivious and one non-oblivious join. The gene query sees a 5x performance gain when the sensitive table is small because the larger oblivious join is replaced with a non-oblivious join. As the sensitive table increases in size, the benefit of join reordering approaches the same level as for the medication query.

9 Related work

9.1 Relevant cryptographic protocols

ORAM. Oblivious RAM [13, 37, 38, 36] is a cryptographic construct for protecting against access pattern leakage. However, ORAM does not fit in Opaque's setting because it has an intrinsically different computation model: serving key-value pairs. We show this problem by devising a simple strawman design using ORAM: put all data items in an in-memory ORAM in Spark.

How can ORAM be utilized if we attempt to sort data, which is an essential operation in SQL? One way to implement sorting on top of ORAM is to simply treat a sorting algorithm's compare-and-swap operation as two ORAM reads and two ORAM writes. This is not viable for three reasons. First, making an ORAM access for each data item is very slow. Second, current ORAM designs are not parallel and distributed, which means that the ORAM accesses will be serialized. Third, we cannot use a regular sorting algorithm because the number of comparisons may be different when run on different underlying data values. This could leak something about the encrypted data and would not provide obliviousness. Therefore, we must use a sorting network anyway, which means that adding ORAM will add an extra polylog(n) factor of accesses.

Other protocols. Fully homomorphic encryption [11, 12] permits computing any function on encrypted data, but is prohibitively slow. Oblivious protocols such as sorting and routing networks [7] are more relevant to Opaque, and Opaque builds on these as discussed in §5.1.

9.2 Non-oblivious systems

A set of database systems encrypt the data so that the service provider cannot see it. These databases can be classified into two types. The first type are encrypted databases, such as CryptDB [33], BlindSeer [31], Monomi [39], AlwaysEncrypted [26], and Seabed [30], that rely on cryptographic techniques for computation. The second type are databases, such as Haven [6], VC3 [34], TrustedDB [5], TDB [23] and GnatDb [40], that require trusted hardware to execute computation.

The main drawback of these systems is that they do not hide access patterns (both in memory and over the network) and hence leak data [41, 28]. Additionally, most of these systems do not fit the distributed analytics setting.

9.3 Oblivious systems

Non-distributed systems. Cipherbase [2] uses trusted hardware to achieve generic functionality for encrypted databases. The base Cipherbase design is not oblivious, but Arasu and Kaushik [3] have proposed oblivious protocols for SQL queries. However, unlike Opaque, their work does not consider the distributed setting. In particular, the proposed oblivious operators are not designed for a parallel setting resulting in sequential execution in Opaque, and do not consider boundary conditions. In addition, Cipherbase's contribution is a design proposal, while Opaque also provides a system and an evaluation.

Ohrimenko et al. [29] provide oblivious algorithms for common ML protocols such as matrix factorization or neural networks, but do not support oblivious relational operators or query optimization. Their focus is not on the distributed setting, and parts of the design (e.g., the choice of a sorting network) and the evaluation focus on single machine performance.

Distributed systems. ObliVM [22] is a platform for generic oblivious computation, and GraphSC [27] is a platform specialized to distributed graph computations built on ObliVM. As we show in §8.3, these systems are three orders of magnitude slower than Opaque. As explained there, they have a different threat model and use different techniques resulting in this higher overhead.

Ohrimenko et al. [28] and M2R [10] provide mechanisms for reducing network traffic analysis leakage for MapReduce jobs. Their solutions do not suffice for Opaque's setting because they do not protect in-memory access patterns. Moreover, they are designed for the simpler setting of a MapReduce job and do not suffice for Opaque's relational operators; further, they do not provide global query optimization of oblivious operators.

10 Conclusion

In this paper, we proposed Opaque, a distributed data analytics platform providing encryption, oblivious computation, and integrity. Opaque contributes a set of distributed oblivious relational operators as well as an oblivious query optimizer. Finally, we show that Opaque is three orders of magnitude faster than state-of-the-art specialized oblivious protocols.

Acknowledgments

We thank the reviewers, the shepherd, Mona Vij and other colleagues from Intel, and Aurojit Panda, for their valuable feedback or discussions. This work was supported by the Intel/NSF CPS-Security grants #1505773 and #20153754, DHS Award HSHQDC-16-3-00083, NSF CISE Expeditions Award CCF-1139158, the UC Berkeley Center for Long-Term Cybersecurity, as well as gifts from Ant Financial, Amazon Web Services, CapitalOne, Ericsson, GE, Google, Huawei, Intel, IBM, Microsoft and VMware.

References

[1] AMPlab, University of California, Berkeley. Big data benchmark. https://amplab.cs.berkeley.edu/benchmark/.

[2] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with Cipherbase. In *6th*, Asilomar, CA, Jan. 2013.

[3] A. Arasu and R. Kaushik. Oblivious query processing. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014.*, pages 26–37, 2014.

[4] M. Armbrust, R. Xin, C. Lian, Y. Huai, D. Liu, J. Bradley, X. Meng, T. Kaftan, M. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, Melbourne, Australia, June 2015.

[5] S. Bajaj and R. Sion. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 205–216, Athens, Greece, June 2011.

[6] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, Oct. 2014.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Sorting networks. In *Introduction to algorithms*, chapter 27. MIT Press, 2001.

[8] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Security Symposium*, Aug. 2016.

[9] H. S. Delugach and T. H. Hinke. Wizard: A database inference analysis and detection system. *IEEE Transactions on Knowledge and Data Engineering*, 8(1):56–66, Feb. 1996.

[10] T. T. A. Dinh, P. Saxena, E.-C. Chang, B. C. Ooi, and C. Zhang. M2r: Enabling stronger privacy in mapreduce computation. In *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[11] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st ACM Symposium on Theory of Computing (STOC)*, Bethesda, MD, May 2009.

[12] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. Cryptology ePrint Archive, Report 2012/099, June 2012.

[13] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

[14] J. E. Gonzalez, D. Crankshaw, A. Dave, R. S. Xin, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, Oct. 2014.

[15] Google Research. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. https://www.tensorflow.org/, 2015.

[16] T. Greene. Biggest data breaches of 2015. Network world. http://www.networkworld.com/article/3011103/security/biggest-data-breaches-of-2015.html, 2015.

[17] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The madlib analytics library: or mad skills, the sql. *Proceedings of the VLDB Endowment*, 5(12):1700–1711, 2012.

[18] T. H. Hinke. Inference aggregation detection in database management systems. In *IEEE Symposium on Security and Privacy*, pages 96–106, 1988.

[19] D. Kaplan, J. Powell, and T. Woller. AMD memory encryption. White paper, Apr. 2016.

[20] T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4):344–354, April 1985.

[21] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. GhostRider: a hardware-software system for memory trace oblivious computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 87–101, 2015.

[22] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. ObliVM: A programming framework for secure computation. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[23] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Oct. 2000.

[24] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[25] X. Meng, J. Bradley, B. Yuvaz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine learning in apache spark. In *Journal of Machine Learning Research, 17(34):1Ð7*, 2016.

[26] Microsoft. Always encrypted database engine. https://msdn.microsoft.com/en-us/library/mt163865.aspx.

[27] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. GraphSC: parallel secure computation made easy. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[28] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Kohlweiss, and D. Sharma. Observing and preventing leakage in mapreduce. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1570–1581. ACM, 2015.

[29] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *Proceedings of the 25th USENIX Security Symposium*, Aug. 2016.

[30] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan. Big data analytics over encrypted datasets with seabed. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[31] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[32] F. Pennic. Anthem suffers the largest healthcare data breach to date. HIT consultant. `http://hitconsultant.net/2015/02/05/anthem-suffers-the-largest-healthcare-data-breach-to-date/`, 2015.

[33] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[34] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 38–54. IEEE, 2015.

[35] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *In Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*, 2017.

[36] E. Stefanov and E. Shi. Multi-cloud oblivious storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 247–258. ACM, 2013.

[37] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. *arXiv preprint arXiv:1106.3652*, 2011.

[38] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: an extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.

[39] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB)*, pages 289–300, Riva del Garda, Italy, Aug. 2013.

[40] R. Vingralek. GnatDb: A small-footprint, secure database system. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, Hong Kong, China, Aug. 2002.

[41] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 640–656. IEEE, 2015.

[42] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, Apr. 2012.

Splinter: Practical Private Queries on Public Data

Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, Matei Zaharia†
MIT CSAIL, †Stanford InfoLab

Abstract

Many online services let users query public datasets such as maps, flight prices, or restaurant reviews. Unfortunately, the queries to these services reveal highly sensitive information that can compromise users' privacy. This paper presents Splinter, a system that protects users' queries on public data and scales to realistic applications. A user splits her query into multiple parts and sends each part to a different provider that holds a copy of the data. As long as any one of the providers is honest and does not collude with the others, the providers cannot determine the query. Splinter uses and extends a new cryptographic primitive called Function Secret Sharing (FSS) that makes it up to an order of magnitude more efficient than prior systems based on Private Information Retrieval and garbled circuits. We develop protocols extending FSS to new types of queries, such as MAX and TOPK queries. We also provide an optimized implementation of FSS using AES-NI instructions and multicores. Splinter achieves end-to-end latencies below 1.6 seconds for realistic workloads including a Yelp clone, flight search, and map routing.

1 Introduction

Many online services let users query large public datasets: some examples include restaurant sites, product catalogs, stock quotes, and searching for directions on maps. In these services, any user can query the data, and the datasets themselves are not sensitive. However, web services can infer a great deal of identifiable and sensitive user information from these queries, such as her current location, political affiliation, sexual orientation, income, etc. [38, 39]. Web services can use this information maliciously and put users at risk to practices such as discriminatory pricing [26, 57, 61]. For example, online stores have charged users different prices based on location [29], and travel sites have also increased prices for certain frequently searched flights [58]. Even when the services are honest, server compromise and subpoenas can leak the sensitive user information on these services [31, 51, 52].

This paper presents Splinter, a system that protects users' queries on public datasets while achieving practical performance for many current web applications. In Splinter, the user divides each query into shares and sends them to different *providers*, which are services hosting a copy of the dataset (Figure 1). As long as any one of the providers is honest and does not collude with the others, the providers cannot discover sensitive information in the query. However, given responses from all the providers, the user can compute the answer to her query.

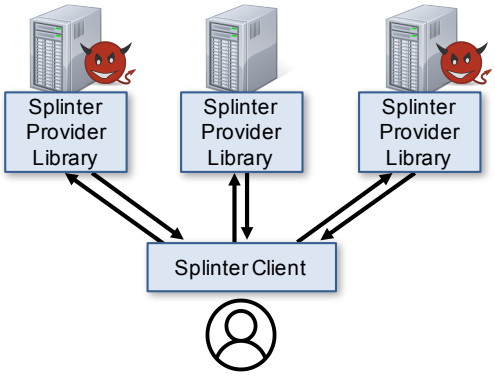


Figure 1: Splinter architecture. The Splinter client splits each user query into shares and sends them to multiple providers. It then combines their results to obtain the final answer. The user's query remains private as long as any one provider is honest.

Previous private query systems have generally not achieved practical performance because they use expensive cryptographic primitives and protocols. For example, systems based on Private Information Retrieval (PIR) [11, 41, 53] require many round trips and high bandwidth for complex queries, while systems based on garbled circuits [8, 32, 64] have a high computational cost. These approaches are especially costly for mobile clients on high-latency networks.

Instead, Splinter uses and extends a recent cryptographic primitive called Function Secret Sharing (FSS) [9, 21], which makes it up to an order of magnitude faster than prior systems. FSS allows the client to split certain functions into shares that keep parameters of the function hidden unless all the providers collude. With judicious use of FSS, many queries can be answered at low CPU and bandwidth cost in only a single network round trip.

Splinter makes two contributions over previous work on FSS. First, prior work has only demonstrated efficient FSS protocols for point and interval functions with additive aggregates such as SUMs [9]. We present protocols that support a more complex set of non-additive aggregates such as MAX/MIN and TOPK at low computational and communication cost. Together, these protocols let Splinter support a subset of SQL that can capture many popular online applications.

Second, we develop an optimized implementation of FSS for modern hardware that leverages AES-NI [56] instructions and multicore CPUs. For example, using the one-way compression functions that utilize modern AES instruction sets, our implementation is 2.5× faster per core than a naïve implementation of FSS. Together, these

optimizations let Splinter query datasets with millions of records at sub-second latency on a single server.

We evaluate Splinter by implementing three applications over it: a restaurant review site similar to Yelp, airline ticket search, and map routing. For all of our applications, Splinter can execute queries in less than 1.6 seconds, at a cost of less than 0.02¢ in server resources on Amazon EC2. Splinter's low cost means that providers could profitably run a Splinter-based service similar to OpenStreetMap routing [46], an open-source maps service, while only charging users a few dollars per month.

In summary, our contributions are:

- Splinter, a private query system for public datasets that achieves significantly lower CPU and communication costs than previous systems.
- New protocols that extend FSS to complex queries with non-additive aggregates, e.g., TOPK and MAX.
- An optimized FSS implementation for modern CPUs.
- An evaluation of Splinter on realistic applications.

2 Splinter Architecture

Splinter aims to protect sensitive information in users' queries from providers. This section provides an overview of Splinter's architecture, security goals, and threat model.

2.1 Splinter Overview

There are two main principals in Splinter: the *user* and the *providers*. Each provider hosts a copy of the data. Providers can retrieve this data from a public repository or mirror site. For example, OpenStreetMap [46] publishes publicly available map, point-of-interest, and traffic data. For a given user query, all the providers have to run it on the same view of the data. Maintaining data consistency from mirror sites is beyond the scope of this paper, but standard techniques can be used [10, 62].

As shown in Figure 1, to issue a query in Splinter, a user splits her query into *shares*, using the Splinter client, and submits each share to a different provider. The user can select any providers of her choice that host the dataset. The providers use their shares to execute the user's query over the cleartext public data, using the Splinter provider library. As long as one provider is *honest* (does not collude with others), the user's sensitive information in the original query remains private. When the user receives the responses from the providers, she combines them to obtain the final answer to her original query.

2.2 Security Goals

The goal of Splinter is to hide sensitive parameters in a user's query. Specifically, Splinter lets users run *parametrized queries*, where both the parameters and query results are hidden from providers. For example, consider the following query, which finds the 10 cheapest flights between a source and destination:

```
SELECT TOP 10 flightid FROM flights
WHERE source = ? AND dest = ?
ORDER BY price
```

Splinter hides the information represented by the questions marks, i.e., the source and destination in this example. The column names being selected and filtered are not hidden. Finally, Splinter also hides the query's results—otherwise, these might be used to infer the source and destination. Splinter supports a subset of the SQL language, which we describe in Section 4.

The easiest way to achieve this property would be for users to download the whole database and run the queries locally. However, this requires substantial bandwidth and computation for the user. Moreover, many datasets change constantly, e.g., to include traffic information or new product reviews. It would be impractical for the user to continuously download these updates. Therefore, our performance objective is to minimize computation and communication costs. For a database of n records, Splinter only requires $O(n \log n)$ computation at the providers and $O(\log n)$ communication (Section 5).

2.3 Threat Model

Splinter keeps the parameters in the user's query hidden as long as at least one of the user-chosen providers does not collude with others. Splinter also assumes these providers are *honest but curious*: a provider can observe the interactions between itself and the client, but Splinter does not protect against providers returning incorrect results or maliciously modifying the dataset.

We assume that the user communicates with each provider through a secure channel (e.g., using SSL), and that the user's Splinter client is uncompromised. Our cryptographic assumptions are standard. We only assume the existence of one-way functions in our two-provider implementation. In our implementation for multiple providers, the security of Paillier encryption [48] is also assumed.

3 Function Secret Sharing

In this section, we give an overview of Function Secret Sharing (FSS), the main primitive used in Splinter, and show how to use it in simple queries. Sections 4 and 5 then describe Splinter's full query model and our new techniques for more complex queries.

3.1 Overview of Function Secret Sharing

Function Secret Sharing [9] lets a client divide a function f into *function shares* $f_1, f_2, \ldots, f_k$ so that multiple parties can help evaluate f without learning certain of its parameters. These shares have the following properties:

- They are close in size to a description of f.
- They can be evaluated quickly (similar in time to f).

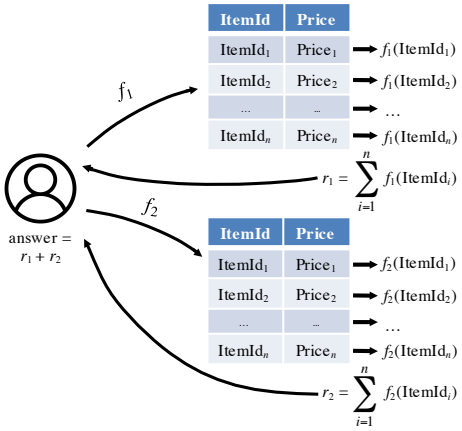


Figure 2: Overview of how FSS can be applied to database records on two providers to perform a COUNT query.

- They sum to the original function f. That is, for any input x, $\sum_{i=1}^{k} f_i(x) = f(x)$. We assume that all computations are done over $\mathbb{Z}_{2^m}$, where m is the number of bits in the output range.
- Given any $k-1$ shares f_i, an adversary cannot recover the parameters of f.

Although it is possible to perform FSS for arbitrary functions [16], practical FSS protocols only exist for *point* and *interval* functions. These take the following forms:

- Point functions f_a are defined as $f_a(x) = 1$ if $x = a$ or 0 otherwise.
- Interval functions are defined as $f_{a,b}(x) = 1$ if $a \le x \le b$ or 0 otherwise.

In both cases, FSS keeps the parameters a and b private: an adversary can tell that it was given a share of a point or interval function, but cannot find a and b. In Splinter, we use the FSS scheme of Boyle et al. [9]. Under this scheme, the shares f_i for both functions require $O(\lambda n)$ bits to describe and $O(\lambda n)$ bit operations to evaluate for a security parameter λ (the size of cryptographic keys), and n is the number of bits in the input domain.

3.2 Using FSS for Database Queries

We can use the additive nature of FSS shares to run private queries over an entire table in addition to a single data record. We illustrate here with two examples.

Example: COUNT query. Suppose that the user wants to run the following query on a table served by Splinter:

```
SELECT COUNT(*) FROM items WHERE ItemId = ?
```

Here, '?' denotes a parameter that the user would like to keep private; for example, suppose the user is searching for `ItemId = 5`, but does not want to reveal this value.

To run this query, the Splinter client defines a point function $f(x) = 1$ if $x = 5$ or 0 otherwise. It then divides this function into function shares $f_1, \ldots, f_n$ and distributes them to the providers, as shown in Figure 2. For simplicity, suppose that there are two providers, who receive shares f_1 and f_2. Because these shares are additive, we know that $f_1(x) + f_2(x) = f(x)$ for every input x. Thus, each provider p can compute $f_p(\text{ItemId})$ for every ItemId in the database table, and send back $r_p = \sum_{i=1}^{n} f_p(\text{ItemId}_i)$ to the client. The client then computes $r_1 + r_2$, which is equal to $\sum_{i=1}^{n} f(\text{ItemId}_i)$, that is, the count of all matching records in the table.

ItemId	Price	f_1(ItemId)	f_2(ItemId)
5	8	10	-9
1	8	3	-3
5	9	10	-9

Figure 3: Simple example table with outputs for the FSS function shares f_1, f_2 applied to the ItemId column. The function is a point function that returns 1 if the input is 5, and 0 otherwise. All outputs are integers modulo 2^m for some m.

To make this more concrete, Figure 3 shows an example table and some sample outputs of the function shares, f_1 and f_2, applied to the ItemId column. There are a few important observations. First, to each provider, the outputs of their function share seem random. Consequently, the provider does not learn the original function f and the parameter "5". Second, because f evaluates to 1 on inputs of 5, $f_1(\text{ItemId}) + f_2(\text{ItemId}) = 1$ for rows 1 and 3. Similarly, $f_1(\text{ItemId}) + f_2(\text{ItemId}) = 0$ for row 2. Therefore, when summed across the providers, each row contributes 1 (if it matches) or 0 (if it does not match) to the final result. Finally, each provider aggregates the outputs of their shares by summing them. In the example, one provider returns 23 to the client, and the other returns -21. The sum of these is the correct query output, 2.

This additivity of FSS enables Splinter to have *low communication costs* for aggregate queries, by aggregating data locally on each provider.

Example: SUM query. Suppose that instead of a COUNT, we wanted to run the following SUM query:

```
SELECT SUM(Price) FROM items WHERE ItemId=?
```

This query can be executed privately with a small extension to the COUNT scheme. As in COUNT, we define a point function f for our secret predicate, e.g., $f(x) = 1$ if $x = 5$ and 0 otherwise. We divide this function into shares f_1 and f_2. However, instead of computing $r_p = \sum_{i=1}^{n} f_p(\text{ItemId}_i)$, each provider p computes

$$r_p = \sum_{i=1}^{n} f_p(\text{ItemId}_i) \cdot \text{Price}_i$$

As before, $r_1 + r_2$ is the correct answer of the query, that is, $\sum_{i=1}^{n} f(\text{ItemId}_i) \cdot \text{Price}_i$. We add in each row's price, Price_i, 0 times if the ItemId is equal to 5, and 1 time if it does not equal 5.

Query format:

SELECT $aggregate_1$, $aggregate_2$, ...
FROM *table*
WHERE *condition*
[GROUP BY $expr_1$, $expr_2$, ...]

aggregate:

- COUNT | SUM | AVG | STDEV (*expr*)
- MAX | MIN (*expr*)
- TOPK (*expr*, *k*, *sort_expr*)
- HISTOGRAM (*expr*, *bins*)

condition:

- *expr* = *secret*
- $secret_1 \leq expr \leq secret_2$
- AND of '=' conditions and up to one interval
- OR of multiple disjoint conditions (e.g., `country="UK" OR country="USA"`)

expr: any public function of the fields in a table row (e.g., `ItemId + 1` or `Price * Tax`)

Figure 4: Splinter query format. The TOPK aggregate returns the top *k* values of *expr* for matching rows in the query, sorting them by *sort_expr*. In conditions, the parameters labeled *secret* are hidden from the providers.

4 Splinter Query Model

Beyond the simple SUM and COUNT queries in the previous section, we have developed protocols to execute a large class of queries using FSS, including non-additive aggregates such as MAX and MIN, and queries that return multiple individual records instead of an aggregate. For all these queries, our protocols are efficient in both computation and communication. On a database of n records, all queries can be executed in $O(n \log n)$ time and $O(\log n)$ communication rounds, and most only require 1 or 2 communication rounds (Figure 6 on page 6).

Figure 4 describes Splinter's supported queries using SQL syntax. Most operators are self-explanatory. The only exception is TOPK, which is used to return up to *k* individual records matching a predicate, sorting them by some expression *sort_expr*. This operator can be used to implement `SELECT...LIMIT` queries, but we show it as a single "aggregate" to simplify our exposition. To keep the number of matching records hidden from providers, the protocol always pads its result to exactly *k* records.

Although Splinter does not support all of SQL, we found it expressive enough to support many real-world query services over public data. We examined various websites, including Yelp, Hotels.com, and Kayak, and found we can support most of their search features as shown in Section 8.1.

Finally, Splinter only "natively" supports fixed-width integer data types. However, such integers can also be used to encode strings and fixed-precision floating point numbers (e.g., SQL DECIMALs). We use them to represent other types of data in our sample applications.

5 Executing Splinter Queries

Given a query in Splinter's query format (Figure 4), the system executes it using the following steps:

1. The Splinter client builds function shares for the condition in the query, as we shall describe in Section 5.1.
2. The client sends the query with all the secret parameters removed to each provider, along with that provider's share of the condition function.
3. If the query has a GROUP BY, each provider divides its data into groups using the grouping expressions; otherwise, it treats the whole table as one group.
4. For each group and each aggregate in the query, the provider runs an evaluation protocol that depends on the aggregate function and on properties of the condition. We describe these protocols in Section 5.2. Some of the protocols require further communication with the client, in which case the provider batches its communication for all grouping keys together.

The main challenge in developing Splinter is designing efficient execution protocols for Splinter's complex conditions and aggregates (Step 4). Our contribution is multiple protocols that can execute non-additive aggregates with low computation and communication costs.

One key insight that pervades our design is that *the best strategy to compute each aggregate depends on properties of the condition function*. For example, if we know that the condition can only match one value of the expression it takes as input, we can simply compute the aggregate's result for *all* distinct values of the expression in the data, and then use a point function to return just one of these results to the client. On the other hand, if the condition can match multiple values, we need a different strategy that can combine results across the matching values. To reason about these properties, we define three *condition classes* that we then use in aggregate evaluation.

5.1 Condition Types and Classes

For any condition c, the Splinter client defines a function f_c that evaluates to 1 on rows where c is true and 0 otherwise, and divides f_c into shares for each provider. Given a condition c, let $E_c = (e_1, \ldots, e_t)$ be the list of expressions referenced in c (the *expr* parameters in its clauses). Because the best strategy for evaluating aggregates depends on c, we divide conditions into three classes:

- *Single-value conditions.* These are conditions that can only be true on one combination of the values of $(e_1, \ldots, e_t)$. For example, conditions consisting of an AND of '=' clauses are single-value.
- *Interval conditions.* These are conditions where the input expressions $e_1, \ldots, e_t$ can be ordered such that c is

true on an interval of the range of values $e_1||e_2||\ldots||e_t$ (where $||$ denotes string concatenation).

- *Disjoint conditions,* i.e., all other conditions.

The condition types described in our query model (Figure 4) can all be converted into sharable functions, and categorized into these classes, as follows:

Equality-only conditions. Conditions of the form $e_1 = secret_1$ AND ... AND $e_t = secret_t$ can be executed as a single point function on the binary string $e_1||\ldots||e_t$. This is simply a point function that can be shared using existing FSS schemes [9]. These conditions are also single-value.

Interval and equality. Conditions of the form $e_1 = secret_1$ AND ... AND $e_{t-1} = secret_{t-1}$ AND $secret_t \leq e_t \leq secret_{t+1}$ can be executed as a single interval function on the binary string $e_1||\ldots||e_t$. This is again supported by existing FSS schemes [9], and is an interval condition.

Disjoint OR. Suppose that $c_1,\ldots,c_t$ are *disjoint* conditions that can be represented using functions $f_{c_1},\ldots,f_{c_t}$. Then $c = c_1$ OR... OR c_t is captured by $f_c = f_{c_1} + \cdots + f_{c_t}$. We share this function across providers by simply giving them shares of the underlying functions f_{c_i}. In the general case, however, c is a disjoint condition where we cannot say much about which inputs give 0 or 1.

5.2 Aggregate Evaluation

5.2.1 Sum-Based Aggregates

To evaluate SUM, COUNT, AVG, STDEV and HISTOGRAM, Splinter sums one or more values for each row regardless of the condition function class. For SUM and COUNT, each provider sums the expression being aggregated or a 1 for each row and multiplies it by $f_i(\text{row})$, its share of the condition function, as in Section 3.2. Computing AVG(x) for an expression x, requires finding SUM(x) and COUNT(x), while computing STDEV(x) requires finding these values and SUM(x^2). Finally, computing a HISTOGRAM into bin boundaries provided by the user simply requires tracking one count per bin, and adding each row's result to the count for its bin. Note that the binning expression is not private—only information about which rows pass the query's condition function.

5.2.2 MAX and MIN

Suppose we are given a query to find MAX(e_0) WHERE $c(e_1,\ldots,e_t)$, for expressions $e_0,\ldots,e_t$. The best evaluation strategy depends on the class of the condition c.

Single-value conditions. If c is only true for one combination of the values $e_1,\ldots,e_t$, each provider starts by evaluating the query

```
SELECT MAX(e0) FROM data GROUP BY e1,...,et
```

This query gives an *intermediate table* with the tuples $(e_1,\ldots,e_t)$ as keys and MAX(e_0) as values. Next, each provider computes $\sum \text{MAX}(e_0)\cdot f_i(e_1,\ldots,e_t)$ across the rows of the intermediate table, where f_i is its share of the condition function. This sum will add a 0 for each non-matching row and MAX(e_0) for the matching row, thus returning the right value. Note that if the original table had n rows, the intermediate table can be built in $O(n)$ time and space using a hash table.

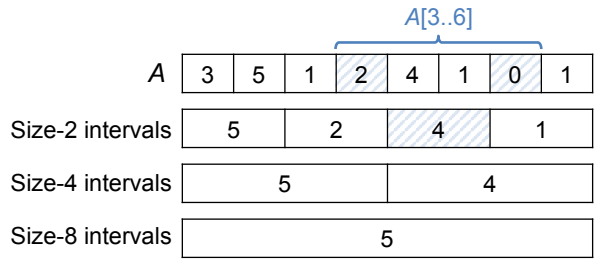


Figure 5: Data structure for querying MAX on intervals. We find the MAX on each power-of-2 aligned interval in the array, of which there are $O(n)$ total. Then, any interval query requires retrieving $O(\log n)$ of these values. For example, to find $MAX(A[3..6])$, we need two size-1 intervals and one size-2.

Interval conditions. Suppose that c is true if and only if $e_1||\ldots||e_t$ is in an interval $[a,b]$, where a and b are secret parameters. As in the single-value case, the providers can build a data structure that helps them evaluate the query without knowing a and b.

In this case, each provider builds an array A of entries (k,v), where the keys are all values of $e_1||\ldots||e_t$ in lexicographic order, and the values are MAX(e_0) for each key. It then computes MAX($A[i..j]$) for all *power-of-2 aligned* intervals of the array A (Figure 5). This data structure is similar to a Fenwick tree [19].

Query evaluation then proceeds in two rounds. First, Splinter counts how many keys in A are less than a and how many are less than b: the client sends the providers shares of the interval functions $k \in [0,a-1]$ and $k \in [0,b-1]$, and the providers apply these to all keys k and return their results. This lets the client find indices i and j in A such that all the keys $k \in [a,b]$ are in $A[i..j]$.

Second, the client sends each provider shares of new point functions that select up to two intervals of size 1, up to two intervals of size 2, etc out of the power-of-2 sized intervals that the providers computed MAXes on, so as to cover exactly $A[i..j]$. Note that any integer interval can be covered using at most 2 intervals of each power of 2. The providers evaluate these functions to return the MAXes for the selected intervals, and the client combines these $O(\log n)$ MAXes to find the overall MAX on $A[i..j]$.[1]

For a table of size n, this protocol requires $O(n\log n)$ time at each provider (to sort the data to build A, and then to answer $O(\log n)$ point function queries). It also

[1] To hide which sizes of intervals were actually required, the client should always request 2 intervals of each size and ignore unneeded ones.

only requires two communication rounds, and $O(\log n)$ communication bandwidth. The same protocol can be used for other associative aggregates, such as products.

Disjoint conditions. If we must find MAX(e_0) WHERE $c(e_1,\ldots,e_t)$ but know nothing about c, Splinter builds an array A of all rows in the dataset sorted by e_0. Finding MAX(e_0) WHERE c is then equivalent to finding the largest index i in A such that $c(A[i])$ is true. To do this, Splinter uses binary search. The client repeatedly sends private queries of the form

```
SELECT COUNT(*) FROM A
WHERE c(e_1,...,e_t) AND index ∈ [secret_1, secret_2],
```

where *index* represents the index of each row in A and the interval for it is kept private. By searching for secret intervals in decreasing power-of-2 sizes, the client can find the largest index i such that $c(A[i])$ is true. For example, if we had an array A of size 8 with largest matching element at $i = 5$, the client would probe $A[0..3]$, $A[4..7]$, $A[4..5]$, $A[6..7]$ and finally $A[4]$ to find that 5 is the largest matching index.

Normally, ANDing the new condition $index \in [secret_1, secret_2]$ with c would cause problems, because the resulting conditions might no longer be in Splinter's supported condition format (ANDs with at most one interval and ORs of disjoint clauses). Fortunately, because the intervals in our condition are always power-of-2 aligned, it can also be written as an equality on the first k bits of *index*. For example, supposing that *index* is a 3-bit value, the condition $index \in [4,5]$ can be written as $index_{0,1} =$ "10", where $index_{0,1}$ is the first two bits of *index*. This lets us AND the condition into all clauses of c.

Once the client has found the largest matching index i, it runs one more query with a point function to select the row with $index = i$. The whole protocol requires $O(\log n)$ communication rounds and $O(n \log n)$ computation and works well if c has many conditions.

However, if c has a small number of OR clauses, an optimization is to run one query for each clause in parallel. The user then resolves the responses locally to find the answer to the original query. Although doing this optimization requires more bandwidth because the returned result size is larger, it avoids the $O(\log n)$ communication rounds and the $O(n \log n)$ computation.

5.2.3 TOPK

Our protocols for evaluating TOPK are similar to those for MAX and MIN. Suppose we are given a query to find TOPK(e, k, e_{sort}) WHERE $c(e_1,\ldots,e_t)$. The evaluation strategy depends on the class of the condition c.

Single-value conditions. If c is only true for one combination of $e_1,\ldots,e_t$, each provider starts by evaluating

Aggregate	Condition	Time	Rounds	Bandwidth
Sum-based	any	$O(n)$	1	$O(1)$
MAX/MIN	1-value	$O(n)$	1	$O(1)$
MAX/MIN	interval	$O(n \log n)$	2	$O(\log n)$
MAX/MIN	disjoint	$O(n \log n)$	$O(\log n)$	$O(\log n)$
TOPK	1-value	$O(n)$	1	$O(1)$
TOPK	interval	$O(n \log n)$	2	$O(\log n)$
TOPK	disjoint	$O(n \log n)$	$O(\log n)$	$O(\log n)$

Figure 6: Complexity of Splinter's query evaluation protocols for a database of size n. For bandwidth, we report the multiplier over the query's normal result size.

```
SELECT TOPK(e, k, e_sort) FROM data
GROUP BY e_1,...,e_t
```

This gives an intermediate table with the tuples $(e_1,\ldots,e_t)$ as keys and TOPK$(\cdot)$ for each group as values, from which we can select the single row matching c as in MAX.

Interval conditions. Here, the providers build the same auxiliary array A as in MAX, storing the TOPK for each key instead. They then compute the TOPKs for power-of-2 aligned intervals in this array. The client finds the interval $A[i..j]$ it needs to query, extracts the top k values for power-of-2 intervals covering it, and finds the overall top k. As in MAX, this protocol requires 2 rounds and $O(\log n)$ communication bandwidth.

Disjoint conditions. Finding TOPK for disjoint conditions is different from MAX because we need to return multiple records instead of just the largest record in the table that matches c. This protocol proceeds as follows:

1. The providers sort the whole table by e_{sort} to create an auxiliary array A.
2. The client uses binary search to find indices i and j in A such that the top k items matching c are in $A[i..j]$. This is done the same way as in MAX, but searching for the largest indices where the count of later items matching c is 0 and k.
3. The client uses a sampling technique (Appendix A) to extract the k records from $A[i..j]$ that match c. Intuitively, although we do not know which rows these are, we build a result table of $> k$ values initialized to 0, and add the FSS share for each row of the data to one row in the result table, chosen by a hash. This scheme extracts all matching records with high probability.

This protocol needs $O(\log n)$ communication rounds and $O(n \log n)$ computation if there are many clauses, but like the protocol for MAX, if the number of clauses in c is small, the user can issue parallel queries for each clause to reduce the communication rounds and computation.

5.3 Complexity

Figure 6 summarizes the complexity of Splinter's query evaluation protocols based on the aggregates and condition classes used. We note that in all cases, the com-

putation time is $O(n \log n)$ and the communication costs are much smaller than the size of the database. This makes Splinter practical even for databases with millions of records, which covers many common public datasets, as shown in Section 8. Finally, the main operations used to evaluate Splinter queries at providers, namely sorting and sums, are highly parallelizable, letting Splinter take advantage of parallel hardware.

6 Optimized FSS Implementation

Apart from introducing new protocols to evaluate complex queries using FSS, Splinter includes an FSS implementation optimized for modern hardware. In this section, we describe our implementation and also discuss how to select the best multi-party FSS scheme for a given query.

6.1 One-Way Compression Functions

The two-party FSS protocol [9] is efficient because of its use of one-way functions. A common class of one-way functions is pseudorandom generators (PRGs) [33], and in practice, AES is the most commonly used PRG because of hardware accelerations, i.e. the AES-NI [56] instruction. Generally, using AES as a PRG is straightforward (use AES in counter mode). However, the use of PRGs in FSS is not only atypical, but it also represents a large portion of the computation cost in the protocol. The FSS protocol requires many instantiations of a PRG with different initial seed values, especially in the two-party protocol [9]. Initializing multiple PRGs with different seed values is very computationally expensive because AES cipher initialization is *much slower* than performing an AES evaluation on an input. Therefore, the challenge in Splinter is to find an efficient PRG for FSS.

Our solution is to use *one-way compression functions*. One way compression functions are commonly used as a primitive in hash functions, like SHA, and are built using a block cipher like AES. In particular, Splinter uses the Matyas-Meyer-Oseas one-way compression function [37] because this function utilizes a *fixed key* cipher. As a result, the Splinter protocol initializes the cipher only once per query.

More precisely, the Matyas-Meyer-Oseas one-way compression function is defined as:

$$F(x) = E_k(x) \oplus x$$

where x is the input, i.e. PRG seed value, and E is a block cipher with a fixed key k.

The output of a one-way compression function is a fixed number of bits, but we can use multiple one-way compression functions with different keys and concatenate the outputs to obtain more bits. Security is preserved because a function that is a concatenation of one-way functions is still a one-way function.

With this one-way compression function, Splinter initializes the cipher, E_k, at the beginning of the query and reuses it for the rest of the query, avoiding expensive AES initialization operations in the FSS protocol. For each record, the Splinter protocol needs to perform only n XORs and n AES evaluations using the AES-NI instruction, where n is the input domain size of the record. In Section 8.3, we show that Splinter's use of one-way compression functions results in a 2.5× speedup over using AES directly as a PRG.

6.2 Selecting the Correct Multi-Party FSS Protocol

There is one efficient protocol for two-party FSS, but for multi-party (more than 2 parties) FSS, there are two different schemes ([9], [14]) that offer different tradeoffs between bandwidth and CPU usage. Both still only require that one provider is honest and does not collude with the remaining providers. In this section, we will provide an overview of the two schemes and discuss their tradeoffs and applicability to different types of applications.

Multi-Party FSS with one-way functions: In [9], the authors present a multi-party protocol based on only one-way functions, which provides good performance. However, there are two limitations. First, the function share size is proportional to the number of parties. Second, the output of the evaluated function share is only additive mod 2 (xor homomorphic), which means that the provider cannot add values locally. This limitation affects queries where there are multiple matches for a condition that requires aggregation, i.e. COUNT and SUM queries. To solve this, the provider responds with all the records that match for a particular user-provided condition, and the client performs the aggregation locally. The size of the result is the largest number of records for a distinct condition, which is usually smaller than the database size. Other queries remain unaffected by this limitation. Applications should use this scheme by default because it provides the fastest response times on low-latency networks. However, for SUM and COUNT queries, an application should be careful using this scheme in settings that are bandwidth-sensitive. Similarly, an application should avoid using this scheme for queries that involve many providers.

Multi-Party FSS with Paillier: In [14], only a point function for FSS is provided, but we modified the scheme to handle interval functions. This scheme has the same additive properties as the two-party FSS protocol in [9], and does not suffer from the limitations of the scheme described above. In fact, the size of the function shares is *independent* of the number of parties. However, this scheme is slower because it uses the Paillier [48] cryptosystem instead of one-way functions. However, it is useful for SUM and COUNT queries in bandwidth-sensitive settings like queries over cellular network, and it is also

beneficial in settings where the user uses many providers.

7 Implementation

We implemented Splinter in C++, using OpenSSL 1.0.2e [45] and the AES-NI hardware instructions for AES encryption. We used GMP [20] for large integers and OpenMP [44] for multithreading. Our optimized FSS library is about 2000 lines of code, and the applications on top of it are about 2000 lines of code. There is around 1500 lines of test code to issue the queries. For comparison, we also implement the multi-party FSS scheme in [14] using 2048 bit Paillier encryption [48]. Our FSS library implementation can be found at `https://github.com/frankw2/libfss`.

8 Evaluation

In our evaluation, we aim to answer one main question: can Splinter be used practically for real applications? To answer this question, we built and evaluated clones of three applications on Splinter: restaurant reviews, flight data, and map routing, using real datasets. We also compare Splinter to previous private systems, and estimate hosting costs. Our providers ran on 64-core Amazon EC2 `x1` servers with Intel Xeon E5-2666 Haswell processors and 1.9 TB of RAM. The client was a 2 GHz Intel Core i7 machine with 8 GB of RAM. Our client's network latency to the providers was 14 ms.

Overall, our experiments show the following:

- Splinter can support realistic applications including the search features of Yelp and flight search sites, and data structures required for map routing.
- Splinter achieves end-to-end latencies below 1.6 sec for queries in these applications on realistic data.
- Splinter's protocols use up to 10× fewer round trips than prior systems and have lower response times.

8.1 Case Studies

Here, we discuss the three application clones we built on Splinter. Figure 7 summarizes our results, and Figure 8 describes the sizes and characteristics of our three datasets. Finally, we also reviewed the search features available in real websites to study how many Splinter supports.

Restaurant review site: We implement a restaurant review site using the Yelp academic dataset [65]. The original dataset contains information for local businesses in 10 cities, but we duplicate the dataset 4 times so that it would approximately represent local businesses in 40 cities. We use the following columns in the data to perform many of the queries expressible on Yelp: name, stars, review count, category, neighborhood and location.

For location-based queries, e.g., restaurants within 5 miles of a user's current location, multiple interval conditions on the longitude and latitude would typically be used. To run these queries faster, we quantize the locations of each restaurant into overlapping hexagons of different radii (e.g., 1, 2 and 5 miles), following the scheme from [40]. We precompute which hexagons each restaurant is in and expose these as additional columns in the data (e.g., `hex1mi` and `hex2mi`). This allows the location queries to use '=' predicates instead of intervals.

For this dataset, we present results for the following three queries:

```
Q1: SELECT COUNT(*) WHERE category="Thai"

Q2: SELECT TOP 10 restaurant
    WHERE category="Mexican" AND
    (hex2mi=1 OR hex2mi=2 OR hex2mi=3)
    ORDER BY stars

Q3: SELECT restaurant, MAX(stars)
    WHERE category="Mexican" OR
    category="Chinese" OR category="Indian"
    OR category="Greek" OR category="Thai"
    OR category="Japanese"
    GROUP BY category
```

Q1 is a count on the number of Thai restaurants. Q2 returns the top 10 Mexican restaurants within a 2 mile radius of a user-specified location by querying three hexagons. We assume that the provider caches the intermediate table for the Top 10 query as described in Section 5.2.3 because it is a common query. Finally, Q3 returns the best rated restaurant from a subset of categories. This requires more communication than other queries because it performs a MAX with many disjoint conditions, as described in Section 5.2.2. Although most queries will probably not have this many disjoint conditions, we test this query to show that Splinter's protocol for this case is also practical.

Flight search: We implement a flight search service similar to Kayak [30], using a public flight dataset [17]. The columns are flight number, origin, destination, month, delay, and price. To find a flight, we search by origin-destination pairs. We present results for two queries:

```
Q1: SELECT AVG(price) WHERE month=3
    AND origin=1 AND dest=2

Q2: SELECT TOP 10 flight_no
    WHERE origin=1 and dest=2 ORDER BY price
```

Q1 shows the average price for a flight during a certain month. Q2 returns the top 10 cheapest flights for a given source and destination, which we encode as integers. Since this is a common query, the results in Figure 7 assume a cached Top 10 intermediate table.

Map routing: We implement a private map routing service, using real traffic map data from [15] for New York City. However, implementing map routing in Splinter is difficult because the providers can perform only a re-

Dataset	Query Desc.	FSS Scheme	Input Bits	Round Trips	Query Size	Response Size	Response Time
Restaurant	COUNT of Thai restaurants (Q1)	Two-party Multi-party	11	1	~2.75 KB ~10 KB	~0.03 KB ~18 KB	57 ms 52 ms
Restaurant	Top 10 Mexican restaurants near user (Q2)	Two-party Multi-party	22	1	~16.5 KB ~1.9 MB	~7 KB ~0.21 KB	150 ms 542 ms
Restaurant	Best rated restaurant in category subset (Q3)	Two-party Multi-party	11	11	~244 KB ~880 KB	~0.7 KB ~396 KB	1.3 s 1.6 s
Flights	AVG monthly price for a certain flight route (Q1)	Two-party Multi-party	17	1	~8.5 KB ~160 KB	~0.06 KB ~300 KB	1.0 s 1.2 s
Flights	Top 10 cheapest flights for a route (Q2)	Two-party Multi-party	13	1	~3.25 KB ~20 KB	~0.3 KB ~0.13 KB	30 ms 39 ms
Maps	Routing query on NYC map	Two-party Multi-party	Grid: 14 Transit Node: 22	2	~12.5 KB ~720 KB	~31 KB ~1.1 KB	1.2 s 1.0 s

Figure 7: Performance of various queries in our case study applications on Splinter. Response times include 14 ms network latency per network round trip. All subqueries are issued in parallel unless they depend on a previous subquery. Query and response sizes are measured per provider. For the multi-party FSS scheme, we run 3 parties. Input bits represent the number of bits in the input domain for FSS, i.e., the maximum size of a column value.

Dataset	# of rows	Size (MB)	Cardinality
Yelp [65]	225,000	23	900 categories
Flights [17]	6,100,000	225	5000 flights
NYC Map [15]	260,000 nodes 733,000 edges	300	1333 transit nodes

Figure 8: Datasets used in the evaluation. The cardinality of queried columns affects the input bit size in our FSS queries.

FSS scheme	Grid	Transit Node	Total
Two Party	0.35 s	0.85 s	1.2 s
Multi-party	0.15 s	0.85 s	1.0 s

Figure 9: Grid, transit node, and total query times for NYC map. A user issues 2 grid queries and one transit node query. The two grid queries are issued together in one message, so there are a total of 2 network round trips.

stricted set of operations. The challenge is to find a shortest path algorithm compatible with Splinter. Fortunately, extensive work has been done to optimize map routing [3]. One algorithm compatible with Splinter is transit node routing (TNR) [2, 5], which has been shown to work well in practice [4]. In TNR, the provider divides up a map into grids, which contain at least one transit node, i.e. a transit node that is part of a "fast" path. There is also a separate table that has the shortest paths between all pairs of transit nodes, which represent a smaller subset of the map. To execute a shortest path query for a given source and destination, the user can use FSS to download the paths in her source and destination grid. She locally finds the shortest path to the source transit node and destination transit node. Finally, she queries the provider for the shortest path between the two transit nodes.

We used the source code from [2] and identified the 1333 transit nodes. We divided the map into 5000 grids, and calculated the shortest path for all transit node pairs. The grid table has 5000 rows representing the edges and nodes in a grid, and the transit node table has about 800,000 rows representing the number of shortest paths for all transit node pairs.

Figure 7 shows the total response time for a routing query between a source and destination in NYC. Figure 9 shows the breakdown of time spent on querying the grid and transit node table. One observation is that the multi-party version is slightly faster than the two party version because it is faster at processing the grid query as shown in Figure 9. The two-party version of FSS requires using GMP operations, which is slower than integer operations used in the multi-party version, but as shown in Figure 7, the two-party version requires much less bandwidth.

Communication costs: Figure 7 shows the total bandwidth of a query request and response for the various case study queries. The sum of those two values represents total bandwidth between the provider and user.

There are two main observations. First, both the query and response sizes are *much smaller* than the size of the database. Second, for non-aggregate queries, the multi-party protocol has a smaller response size compared to the two-party protocol but the query size is much larger than the two-party protocol, leading to higher overall communication. For aggregate queries, in Section 6.2, we mention that the faster multi-party FSS scheme is only xor homomorphic, so it outputs all the matches for a specific predicate. The user has to perform the aggregation locally, leading to a larger response size than the two-party protocol. Overall, the multi-party protocols have higher total bandwidth compared to the two-party protocols despite some differences in response size.

Website	Search Feature	Splinter Primitive
Yelp	Booking Method, Cities, Distance	Equality
	Price	Range
	Best Match, Top Rated, Most Reviews	Sorting
	Free text search	—
Hotels.com	Destination, Room type, Amenities	Equality
	Check in/out, Price, Ratings	Range
	Stars, Distance, Ratings, Price	Sorting
	Name contains	—
Kayak	From/To, Cabin, Passengers, Stops	Equality
	Date, Flight time, Layover time, Price	Range
Google Maps	From/To, Transit type, Route options	Equality

Figure 10: Example website search features and their equivalent Splinter query class.

Coverage of supported queries: We also manually characterized the applicability of Splinter to several widely used online services by studying how many of the search fields on these services' interfaces Splinter can support. Figure 10 shows the results. Most services use equality and range predicates: for example, the Hotels.com user interface includes checkboxes for selecting categories, neighborhoods, stars, etc, a range fields for price, and one free-text search field that Splinter does not support. In general, all features except free-text search could be supported by Splinter. For free-text search, simple keywords that map to a category (e.g., "grocery store") could also be supported.

8.2 Comparison to Other Private Query Systems

To the best of our knowledge, the most recent private query system that can perform a similar class of queries as Splinter is that of Olumofin et al. [41], which uses multi-party PIR. Olumofin et al. creates an m-ary ($m = 4$) B+ index tree for the dataset and uses PIR to search through it to return various results. As a result, their queries require $O(\log_m n)$ round trips, where n is the number of records. In Splinter, the number of rounds trips does not depend on the size of the database for most queries. As shown in Section 5.2.2 and Section 5.2.3, the exception is for MIN/MAX and TOPK queries with many disjoint conditions where Splinter's communication is similar; if there are a small number of disjoint conditions, Splinter will be faster than previous systems because the user can issue parallel queries.

Figure 11 shows the round trips required in Olumofin et al.'s system and in Splinter for the queries in our case studies. Splinter improves over [41] by up to an order of magnitude. Restaurant Q3 uses a disjoint MAX, so the communication is similar.

We see a similar performance difference from the results in [41]'s evaluation, which reports response times of of 2-18 seconds for queries with several million records, compared to 50 ms to 1.6 seconds in Splinter. Moreover, the experiments in [41] do not use a real network, despite having a large number of round trips, so their response times would be even longer on high-latency networks. Finally, the system in [41] has weaker security guarantees: it requires *all* the providers to be honest, whereas Splinter only requires that *one* provider is honest.

Splinter Query	RTs in [41]	RTs in Splinter
Restaurant Q1	10	1
Restaurant Q2	6	1
Restaurant Q3	6	11
Flights Q1	13	1
Flights Q2	8	1
Map Routing	19	2

Figure 11: For our queries, we show the round trips required for the system of Olumofin et al. [41] and Splinter.[2]

For maps, a recent system by Wu et al. [64] used garbled circuits for map routing. They achieve response times of 140-784 seconds for their maps with Los Angeles as their largest map, and require 8-16 MB of total bandwidth. Splinter has a response time of 1.2 seconds on a larger map (NYC), which is 100× lower, and with a total bandwidth of 45-725 KB, which is 10× lower.

8.3 FSS Microbenchmarks

Cryptographic operations are the main cost in Splinter. We present microbenchmarks to show these costs of various parts of the FSS protocol, tradeoffs between various FSS protocols, and the throughput of FSS. The microbenchmarks also show why the response times in Figure 7 are different between the two-party and multi-party FSS cases. All of these experiments are done on one core to show the per-core throughput of the FSS protocol.

Two-party FSS: For two-party FSS, generating a function share takes less than 1 ms. The speed of FSS evaluation is proportional to the size of the input domain, i.e. number of bits per record. We can perform around 700,000 FSS evaluations per second on 24-bit records, i.e. process around 700,000 distinct 24-bit records, using one-way compression functions. Figure 12 shows the per-core throughput of our implementation for different FSS schemes, i.e. number of unique database records that can be processed per second. It also shows that using one-way compression functions as described in Section 6, we obtain a 2.5× speedup over using AES as a PRG.

Multi-party FSS: As shown in Figure 12, for the multi-party FSS scheme from [9] that only uses one-way functions, the time to generate the function share and evaluate it is proportional to $2^{n/2}$ where n is the number of bits in

[2] The number of round trips in Restaurant Q3 is $O(\log n)$ in both Splinter and Olumofin et al., but the absolute number is higher in Splinter because we use a binary search whereas Olumofin et al. use a 4-ary tree. Splinter could also use a 4-ary search to achieve the same number of round trips, but we have not yet implemented this.

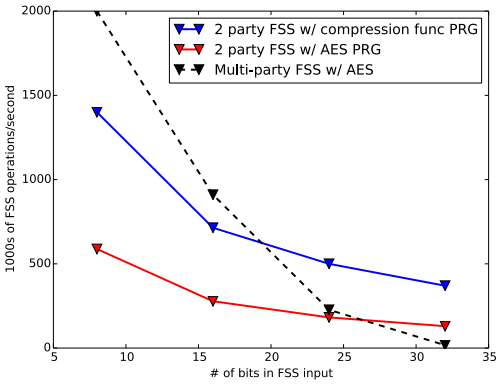


Figure 12: Per-core throughput of various FSS protocols. The graph shows the number of FSS operations that can be performed, i.e. database records processed, per second for various input sizes, on one core.

Time to generate function shares

# of bits	Query Gen in Boyle et al [9]	Query Gen in Riposte [14]
8	< 1 ms	0.06 s
16	< 1 ms	1 s
24	44 ms	16 s
32	166 ms	265 s

Figure 13: Query generation times for multi-party FSS schemes using one-way functions [9] and using Paillier [14].

the input domain. The size of the share scales with $2^{n/2}$ rather than just n in the two-party case. An important observation is that using one-way compression functions instead of AES does not make a significant difference for multi-party FSS because the PRG is called less often compared to two-party FSS. For small input domains ($<$ 20 bits), the multi-party version of FSS is faster than the 2-party version, but as stated in Section 6.2, a provider cannot aggregate locally for SUM and COUNT queries.

For the scheme from [14], which uses Paillier encryption, generating a function share is slower compared to [9] because it requires many exponentiations over a large integer group and depends on the number of record bits. Figure 13 shows a summary of the query generation times for both schemes. However, the evaluation of the function share is independent of the size of the input domain. The output of the function share in [14] returns a group element that is additive in a large integer group, but in order to have this property, the performance is lower compared to [9]. We can perform 250 FSS evaluations a second, but this lower performance is useful for SUM and COUNT operations on bandwidth-constrained clients.

8.4 Hosting Costs

We estimate Splinter's server-side computation cost on Amazon EC2, where the cost of a CPU-hour is about 5 cents [1]. We found that most of our queries cost less than 0.002¢. Map queries are a bit more costly, about 0.02¢ to run a shortest path query for NYC, because the amount of computation required is higher.

9 Discussion and Limitations

Economic feasibility: Although it is hard to predict real-world deployment, we believe that Splinter's low cost makes it economically feasible for several types of applications. Studies have shown that many consumers are willing to pay for services that protect their privacy [28, 55]. In fact, users might not use certain services because of privacy concerns [52, 54]. Well-known sites like OkCupid, Pandora, Youtube, and Slashdot allow users to pay a monthly fee to remove ads that collect their information, showing there is already a demographic willing to pay for privacy. As shown in Section 8.4, the cost of running queries on Splinter is low, with our most expensive query, map routing, costing less than 0.02¢ in AWS resources. At this cost, providers could offer Splinter-based map routing for a subscription fee of $1 per month, assuming each user makes 100 map queries per day. Splinter's trust model, where only one provider needs to be honest, also makes it easy for new providers to join the market, increasing users' privacy. Whether such a business model would work in practice is beyond the scope of this paper.

One obstacle to Splinter's use is that many current data providers, such as Yelp and Google Maps, generate revenue primarily by showing ads and mining user data. Nonetheless, there are already successful open databases containing most of the data in these services, such as OpenStreetMap [46], and basic data on locations does not change rapidly once collected. Moreover, the availability of techniques like Splinter might make it easier to introduce regulation about privacy in certain settings, similar to current privacy regulations in HIPAA [27].

Unsupported queries: As shown in Section 4, Splinter supports only a subset of SQL. Splinter does not support partial text matching or image matching, which are common in types of applications that might use Splinter. Moreover, Splinter cannot support private joins, i.e. Splinter can only support joining with another table if the join condition is public. Despite these limitations, our study in Section 8.1 shows Splinter can support many application search interfaces.

Number of providers: One limitation of Splinter is that a Splinter-based service has to be deployed on at least two providers. However, previous PIR systems described in Section 10 also require at least two providers. Unlike those systems, Splinter requires only *one* honest provider whereas those systems require *all* providers be honest. Moreover, current multi-party FSS schemes do not scale well past three providers, but we believe that further research will improve its efficiency.

Full table scans: FSS, like PIR, requires scanning the whole input dataset on every Splinter query, to prevent providers from figuring out which records have been accessed. Despite this limitation, we have shown that Splinter is practical on large real-world datasets, such as maps.

Splinter needs to scan the whole table only for conditions that contain sensitive parameters. For example, consider the query:

```
SELECT flight from table WHERE src=SFO
AND dst=LGA AND delay < 20
```

If the user does not consider the delay of 20 in this query to be private, Splinter could send it in the clear. The providers can then create an intermediate table with only flights where the delay < 20 and apply the private conditions only to records in this table. In a similar manner, users querying geographic data may be willing to reveal their location at the country or state level but would like to keep their location inside the state or country private.

Maintaining consistent data views: Splinter requires that each provider executes a given user query on the same copy of the data. Much research in distributed systems has focused on ensuring databases consistency across multiple providers [13, 43, 63]. Using the appropriate consistency techniques is dependent on the application and an active area of research. Applying those techniques in Splinter is beyond the scope of this paper.

10 Related Work

Splinter is related to work in Private Information Retrieval (PIR), garbled circuit systems, encrypted data systems, and Oblivious RAM (ORAM) systems. Splinter achieves higher performance than these systems through its mapping of database queries to the Function Secret Sharing (FSS) primitive.

PIR systems: Splinter is most closely related to systems that use Private Information Retrieval (PIR) [12] to query a database privately. In PIR, a user queries for the i^{th} record in the database, and the database does not learn the queried index i or the result. Much work has been done on improving PIR protocols [42, 47]. Work has also been done to extend PIR to return multiple records [24], but it is computationally expensive. Our work is most closely related to the system in [41], which implements a parametrized SQL-like query model similar to Splinter using PIR. However, because this system uses PIR, it has up to $10\times$ more round trips and much higher response times for similar queries.

Popcorn [25] is a media delivery service that uses PIR to hide user consumption habits from the provider and content distributor. However, Popcorn is optimized for streaming media databases, like Netflix, which have a small number (about 8000) of large records.

The systems above have a weaker security model: *all* the providers need to be honest. Splinter only requires *one* honest provider, and it is more practical because it extends Function Secret Sharing (FSS) [9, 21], which lets it execute complex operations such as sums in one round trip instead of only extracting one data record at a time.

Garbled circuits: Systems such as Embark [32], BlindBox [59], and private shortest path computation systems [64] use garbled circuits [7, 22] to perform private computation on a single untrusted server. Even with improvements in practicality [6], these techniques still have high computation and bandwidth costs for queries on large datasets because a new garbled circuit has to be generated for each query. (Reusable garbled circuits [23] are not yet practical.) For example, the recent map routing system by Wu et al. [64] uses garbled circuits and has $100\times$ higher response time and $10\times$ higher bandwidth cost than Splinter.

Encrypted data systems: Systems that compute on encrypted data, such as CryptDB [49], Mylar [50], SPORC [18], Depot [36], and SUNDR [34], all try to protect private data against a server compromise, which is a different problem than what Splinter tries to solve. CryptDB is most similar to Splinter because it allows for SQL-like queries over encrypted data. However, all these systems protect against a single, potentially compromised server where the user is storing data privately, but they do not hide data access patterns. In contrast, Splinter hides data access patterns and a user's query parameters but is only designed to operate on a public dataset that is hosted at multiple providers.

ORAM systems: Splinter is also related to systems that use Oblivious RAM [35, 60]. ORAM allows a user to read and write data on an untrusted server without revealing her data access patterns to the server. However, ORAM cannot be easily applied into the Splinter setting. One main requirement of ORAM is that the user can only read data that she has written. In Splinter, the provider hosts a public dataset, not created by any specific user, and many users need to access the same dataset.

11 Conclusion

Splinter is a new private query system that protects sensitive parameters in SQL-like queries while scaling to realistic applications. Splinter uses and extends a recent cryptography primitive, Function Secret Sharing (FSS), allowing it to achieve up to an order of magnitude better performance compared to previous private query systems. We develop protocols to execute complex queries with low computation and bandwidth. As a proof of concept, we have evaluated Splinter with three sample applications—a Yelp clone, map routing, and flight search—and showed that Splinter has low response times from 50 ms to 1.6 seconds with low hosting costs.

Acknowledgements

We thank our anonymous reviewers and our shepherd Tom Anderson for their useful feedback. We would also like to thank James Mickens, Tej Chajed, Jon Gjengset, David Lazar, Malte Schwarzkopf, Amy Ousterhout, Shoumik Palkar, and Peter Bailis for their comments. This work was partially supported by an NSF Graduate Research Fellowship (Grant No. 2013135952), NSF awards CNS-1053143, CNS-1413920, CNS-1350619, CNS-1414119, Alfred P. Sloan Research Fellowship, Microsoft Faculty Fellowship, an Analog Devices grant, SIMONS Investigator award Agreement Dated 6-5-12, and VMware.

References

[1] Amazon. Amazon EC2 Instance Pricing. `https://aws.amazon.com/ec2/pricing/`.

[2] J. Arz, D. Luxen, and P. Sanders. Transit node routing reconsidered. In *Experimental Algorithms*, pages 55–66. 2013.

[3] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. *arXiv preprint arXiv:1504.05140*, 2015.

[4] H. Bast, S. Funke, and D. Matijevic. Ultrafast shortest-path queries via transit nodes. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, 74:175–192, 2009.

[5] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566–566, 2007.

[6] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, pages 478–492, San Francisco, CA, May 2013.

[7] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 784–796, Raleigh, NC, Oct. 2012.

[8] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: A system for secure multi-party computation. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, pages 257–266, Alexandria, VA, Oct. 2008.

[9] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing. In *Proceedings of the 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 337–367. Sofia, Bulgaria, Apr. 2015.

[10] C.-H. Chi, C.-K. Chua, and W. Song. A novel ownership scheme to maintain web content consistency. In *International Conference on Grid and Pervasive Computing*, pages 352–363. Springer, 2008.

[11] B. Chor, N. Gilboa, and M. Naor. *Private information retrieval by keywords*. Citeseer, 1997.

[12] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

[13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[14] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An anonymous messaging system handling millions of users. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, San Jose, CA, May 2015.

[15] DIMACS. 9th DIMACS Implementation Challenge - Shortest Paths. `http://www.dis.uniroma1.it/challenge9/download.shtml`.

[16] Y. Dodis, S. Halevi, R. Rothblum, and D. Wichs. Spooky encryption and its applications. In *Proceedings of the 36th Annual International Cryptology Conference (CRYPTO)*, pages 93–122, Santa Barbara, CA, Aug. 2016.

[17] Enigma. Arrival Data for Non-Stop Domestic Flights by Major Air Carriers for 2012. `https://app.enigma.io/table/us.gov.dot.rita.transstats.on-time-performance.2012`.

[18] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[19] P. M. Fenwick. A new data structure for cumulative frequency tables. *Software – Practice and Experience*, 24(3):327–336, Mar. 1994.

[20] F. S. Foundation. GNU Multi Precision Arithmetic Library. `https://gmplib.org/`.

[21] N. Gilboa and Y. Ishai. Distributed point functions and their applications. In *Proceedings of the 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 640–658. Copenhagen, Denmark, May 2014.

[22] S. Goldwasser. Multi party computations: past and present. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PDC)*, pages 1–6, 1997.

[23] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC)*, pages 555–564, Palo Alto, CA, June 2013.

[24] J. Groth, A. Kiayias, and H. Lipmaa. Multi-query computationally-private information retrieval with constant communication rate. In *International Workshop on Public Key Cryptography*, pages 107–123. Springer, 2010.

[25] T. Gupta, N. Crooks, S. T. Setty, L. Alvisi, and M. Walfish. Scalable and private media consumption with Popcorn. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 91–107, Santa Clara, CA, Mar. 2016.

[26] A. Hannak, G. Soeller, D. Lazer, A. Mislove, and C. Wilson. Measuring price discrimination and steering on e-commerce web sites. In *Proceedings of the Conference on Internet Measurement Conference (IMC)*, pages 305–318, 2014.

[27] Health Insurance Portability and Accountability Act. `https://en.wikipedia.org/wiki/Health_Insurance_Portability_and_Accountability_Act`.

[28] D. Indiviglio. Most Internet Users Willing to Pay for Privacy, December 22 2010. `http://www.theatlantic.com/business/archive/2010/12/most-internet-users-willing-to-pay-for-privacy/68443/`.

[29] J. S.-V. Jennifer Valentino-Devries and A. Soltani. Websites Vary Prices, Deals Based on Users' Information, December 24 2012. Wall Street Journal.

[30] Kayak. Kayak. `https://www.kayak.com`.

[31] J. Kincaid. Another Security Hole Found on Yelp, Facebook Data Once Again Put at Risk, May 11 2010. `http://techcrunch.com/2010/05/11/another-security-hole-found-on-yelp-facebook-data-once-again-put-at-risk/`.

[32] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu. Embark: Securely outsourcing middleboxes to the cloud. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 255–273, Santa Clara, CA, Mar. 2016.

[33] L. A. Levin. One way functions and pseudorandom generators. *Combinatorica*, 7(4):357–363, 1987.

[34] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–106, San Francisco, CA, Dec. 2004.

[35] J. R. Lorch, B. Parno, J. Mickens, M. Raykova, and J. Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 199–213, San Jose, CA, Feb. 2013.

[36] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[37] S. M. Matyas, C. H. Meyer, and J. Oseas. Generating strong one-way functions with cryptographic algorithms. *IBM Technical Disclosure Bulletin*, 27(10A):5658–5659, 1985.

[38] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*, pages 111–125, Oakland, CA, May 2008.

[39] A. Narayanan and V. Shmatikov. Myths and fallacies of personally identifiable information. *Communications of the ACM*, 53(6):24–26, 2010.

[40] A. Narayanan, N. Thiagarajan, M. Lakhani, M. Hamburg, and D. Boneh. Location privacy via private proximity testing. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2011.

[41] F. Olumofin and I. Goldberg. Privacy-preserving queries over relational databases. In *Proceedings of the 10th Privacy Enhancing Technologies Symposium*, pages 75–92, Berlin, Germany, 2010.

[42] F. Olumofin and I. Goldberg. Revisiting the computational practicality of private information retrieval. In *Financial Cryptography and Data Security*, pages 158–172. 2011.

[43] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 305–319, Philadelphia, PA, June 2014.

[44] OpenMP. OpenMP. `http://www.openmp.org/`.

[45] OpenSSL. OpenSSL. `https://openssl.org`.

[46] OpenStreetMap. OpenStreetMap. `https://www.openstreetmap.org/`.

[47] R. Ostrovsky and W. E. Skeith III. A survey of single-database private information retrieval: Techniques and applications. In *Public Key Cryptography (PKC)*, pages 393–411. 2007.

[48] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the 18th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 223–238, Prague, Czech Republic, May 1999.

[49] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100, Cascais, Portugal, Oct. 2011.

[50] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan. Building web applications on top of encrypted data using Mylar. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 157–172, Seattle, WA, Apr. 2014.

[51] F. Y. Rashid. Twitter Breached, Attackers Stole 250,000 User Data, February 2 2013. `http://securitywatch.pcmag.com/none/307708-twitter-breached-attackers-stole-250-000-user-data`.

[52] R. Ravichandran, M. Benisch, P. G. Kelley, and N. M. Sadeh. Capturing social networking privacy preferences. In *Proceedings of the 9th Privacy Enhancing Technologies Symposium*, pages 1–18, Seattle, WA, Aug. 2009.

[53] J. Reardon, J. Pound, and I. Goldberg. Relational-complete private information retrieval. *University of Waterloo, Tech. Rep. CACR*, 34:2007, 2007.

[54] P. F. Riley. The Tolls of Privacy: An underestimated roadblock for electronic toll collection usage. *Computer Law & Security Review*, 24(6):521–528, 2008.

[55] R. J. Rosen. Study: Consumers Will Pay $5 for an App that Respects their Privacy, December 26 2013. http://www.theatlantic.com/technology/archive/2013/12/study-consumers-will-pay-5-for-an-app-that-respects-their-privacy/282663/.

[56] J. Rott. Intel advanced encryption standard instructions (AES-NI). Technical report, Technical report, Intel, 2010.

[57] F. Salmon. Why the Internet is Perfect for Price Discrimination, September 3 2013. http://blogs.reuters.com/felix-salmon/2013/09/03/why-the-internet-is-perfect-for-price-discrimination/.

[58] R. Seaney. Do Cookies Really Raise Airfares?, April 30 2013. http://www.usatoday.com/story/travel/columnist/seaney/2013/04/30/airfare-expert-do-cookies-really-raise-airfares/2121981/.

[59] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM SIGCOMM*, pages 213–226, London, United Kingdom, Aug. 2015.

[60] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Nov. 2013.

[61] A. Tanner. Different customers, Different prices, Thanks to big data, March 21 2014. http://www.forbes.com/sites/adamtanner/2014/03/26/different-customers-different-prices-thanks-to-big-data/.

[62] R. Tewari, T. Niranjan, and S. Ramamurthy. WCDP: A protocol for web cache consistency. In *Proceedings of the 7th Web Caching Workshop*. Citeseer, 2002.

[63] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, Nov. 2013.

[64] D. J. Wu, J. Zimmerman, J. Planul, and J. C. Mitchell. Privacy-preserving shortest path computation. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2016.

[65] Yelp. Yelp Academic Dataset. https://www.yelp.com/dataset_challenge/dataset.

A Extracting Disjoint Records with FSS

This appendix describes our sampling-based technique for returning multiple records using FSS, used in TOPK queries with disjoint conditions (Section 5.2.3). Given a table T of records and a condition c that matches up to k records, we wish to return those records to the client with high probability without revealing c.

To solve this problem, the providers each create a result table R of size $l > k$, containing (value, count) columns all initialized to 0. They then iterate through the records and choose a result row to update for each record based on a hash function h of its index i. For each record r, each provider adds $1 \cdot f_c(r)$ to $R[h(i)]$.count and $r \cdot f_c(r)$ to $R[h(i)]$.value, where f_c is its share of the condition c. The client then adds up the R tables from all the providers to build up a single table, which contains a value and count for all indices that a record matching c hashed into.

Given this information, the client can tell how many records hashed into each index: entries with count=1 have only one record, which can be read from the entry's value. Unfortunately, entries with higher counts hold multiple records that were added together in the value field. To recover these entries, the client can run the same process multiple times in parallel with different hash functions h.

In general, for any given value of r and k, the probability of a given record colliding with another under each hash function is a constant (e.g., it is less than $1/3$ for $r = 3k$). Repeating this process with more hash functions causes the probability to fall exponentially. Thus, for any k, we can return all the distinct results with high probability using only $O(\log k)$ hash functions and hence only $O(\log k)$ extra communication bandwidth.

VFP: A Virtual Switch Platform for Host SDN in the Public Cloud

Daniel Firestone, *Microsoft*

Abstract

Many modern scalable cloud networking architectures rely on host networking for implementing VM network policy - e.g. tunneling for virtual networks, NAT for load balancing, stateful ACLs, QoS, and more. We present the Virtual Filtering Platform (VFP) - a programmable virtual switch that powers Microsoft Azure, a large public cloud, and provides this policy. We define several major goals for a programmable virtual switch based on our operational experiences, including support for multiple independent network controllers, policy based on connections rather than only on packets, efficient caching and classification algorithms for performance, and efficient offload of flow policy to programmable NICs, and demonstrate how VFP achieves these goals. VFP has been deployed on >1M hosts running IaaS and PaaS workloads for over 4 years. We present the design of VFP and its API, its flow language and compiler used for flow processing, performance results, and experiences deploying and using VFP in Azure over several years.

1. Introduction

The rise of public cloud workloads, such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform [13-15], has created a new scale of datacenter computing, with vendors regularly reporting server counts in the millions. These vendors not only have to provide scale and the high density/performance of Virtual Machines (VMs) to customers, but must provide rich network semantics, such as private virtual networks with customer supplied address spaces, scalable L4 load balancers, security groups and ACLs, virtual routing tables, bandwidth metering, QoS, and more.

This policy is sufficiently complex that it often cannot economically be implemented at scale in traditional core routers and hardware. Instead a common approach has been to implement this policy in software on the VM hosts, in the virtual switch (vswitch) connecting VMs to the network, which scales well with the number of servers, and allows the physical network to be simple, scalable and very fast. As this model separates a centralized control plane from a data plane on the host, it is widely considered an example of Software Defined Networking (SDN) – in particular, host-based SDN.

As a large public cloud provider, Azure has built its cloud network on host based SDN technologies, using them to implement almost all virtual networking features we offer. Much of the focus around SDN in recent years has been on building scalable and flexible network controllers and services, which is critical. However, the design of the programmable vswitch is equally important. It has the dual and often conflicting requirements of a highly programmable dataplane, with high performance and low overhead, as cloud workloads are cost and performance sensitive.

In this paper, we present the Virtual Filtering Platform, or VFP – our cloud scale virtual switch that runs on all of our hosts. VFP is so named because it acts as a filtering engine for each virtual NIC of a VM, allowing controllers to program their SDN policy. Our goal is to present both our design and our experiences running VFP in production at scale, and lessons we learned.

1.1 Related Work

Throughout this paper, we use two motivating examples from the literature and demonstrate how VFP supports their policies and actions. The first is VL2 [2], which can be used to create virtual networks (VNETs) on shared hardware using stateless tunneling between hosts. The second is Ananta [4], a scalable Layer-4 load balancer, which scales by running the load balancing NAT in the vswitch on end hosts, leaving the in-network load balancers stateless and scalable.

In addition, we make references and comparisons to OpenFlow [5], a programmable forwarding plane protocol, and OpenVswitch [1] (OVS), a popular open source vswitch implementing OpenFlow. These are two seminal projects in the SDN space. We point out core design differences from the perspective of a public cloud on how our constraints can differ from those of open source projects. It is our goal to share these learnings with the broader community.

2. Design Goals and Rationale

VFP's design has evolved over time based on our experiences running a large public cloud platform. VFP was not our original vswitch, nor were its original functions novel ideas in host networking – VL2 and Ananta already pioneered such use of vswitches.

Originally, we built networking filter drivers on top of Windows's Hyper-V hypervisor for each host function, which we chained together in a vswitch – a stateful firewall driver for ACLs, a tunneling driver for VL2 VNETs, a NAT driver for Ananta load balancing, a QoS driver, etc. As host networking became our main tool for virtualization policy, we decided to create VFP in 2011 after concluding that building new fixed filter drivers for host networking functions was not scalable

or desirable. Instead, we created a single platform based on the Match-Action Table (MAT) model popularized by projects such as OpenFlow [5]. This was the origin of our VFPAPI programming model for VFP clients.

VFP's core design goals were taken from lessons learned in building and running both these filters, and the network controllers and agents on top of them.

2.1 Original Goals

The following were founding goals of the VFP project:

1. *Provide a programming model allowing for multiple simultaneous, independent network controllers to program network applications, minimizing cross-controller dependencies.*

Implementations of OpenFlow and similar MAT models often assume a single distributed network controller that owns programming the switch (possibly taking input from other controllers). Our experience is that this model doesn't fit cloud development of SDN – instead, independent teams often build new network controllers and agents for those applications. This model reduces complex dependencies, scales better and is more serviceable than adding logic to existing controllers. We needed a design that not only allows controllers to independently create and program flow tables, but would enforce good layering and boundaries between them (e.g. disallow rules to have arbitrary GOTOs to other tables, as in OpenFlow) so that new controllers could be developed to add functionality without old controllers needing to take their behavior into account.

2. *Provide a MAT programming model capable of using connections as a base primitive, rather than just packets – stateful rules as first class objects.*

OpenFlow's original MAT model derives historically from programming switching or routing ASICs, and so assumes that packet classification must be stateless due to hardware resources available. However, we found our controllers required policies for connections, not just packets – for example end users often found it more useful to secure their VMs using stateful Access Control Lists (ACLs) (e.g. allow outbound connections, but not inbound) rather than stateless ACLs used in commercial switches. Controllers also needed NAT (e.g. Ananta) and other stateful policies. Stateful policy is more tractable in soft switches than in ASIC ones, and we believe our MAT model should take advantage of that.

3. *Provide a programming model that allows controllers to define their own policy and actions, rather than implementing fixed sets of network policies for predefined scenarios.*

Due to limitations of the MAT model provided by OpenFlow (historically, a limited set of actions, limited rule scalability and no table typing), OpenFlow switches such as OVS have added virtualization functionality outside of the MAT model. For example, constructing virtual networks is accomplished via a virtual tunnel endpoint (VTEP) schema [29] in OVSDB [8], rather than rules specifying which packets to encapsulate (encap) and decapsulate (decap) and how to do so.

We prefer instead to base all functionality on the MAT model, trying to push as much logic as possible into the controllers while leaving the core dataplane in the vswitch. For instance, rather than a schema that defines what a VNET is, a VNET can be implemented using programmable encap and decap rules matching appropriate conditions, leaving the definition of a VNET in the controller. We've found this greatly reduces the need to continuously extend the dataplane every time the definition of a VNET changes.

The P4 language [9] attempts a similar goal for switches or vswitches [38], but is very generic, e.g. allowing new headers to be defined on the fly. Since we update our vswitch much more often than we add new packet headers to our network, we prefer the speed of a library of precompiled fast header parsers, and a language structured for stateful connection-oriented processing.

2.2 Goals Based on Production Learnings

Based on lessons from initial deployments of VFP, we added the following goals for VFPv2, a major update in 2013-14, mostly around serviceability and performance:

4. *Provide a serviceability model allowing for frequent deployments and updates without requiring reboots or interrupting VM connectivity for stateful flows, and strong service monitoring.*

As our scale grew dramatically (from O(10K) to O(1M) hosts), more controllers built on top of VFP, and more engineers joined us, we found more demand than ever for frequent updates, both features and bug fixes. In Infrastructure as a Service (IaaS) models, we also found customers were not tolerant of taking downtime for individual VMs for updates. This goal was more challenging to achieve with our complex stateful flow model, which is nontrivial to maintain across updates.

5. *Provide very high packet rates, even with a large number of tables and rules, via extensive caching.*

Over time we found more and more network controllers being built as the host SDN model became more popular, and soon we had deployments with large numbers of flow tables (10+), each with many rules, reducing performance as packets had to traverse each table. At the same time, VM density on hosts was increasing, pushing us from 1G to 10G to 40G and even faster NICs. We needed to find a way to scale to more policy without impacting performance, and concluded we needed to perform compilation of flow actions

across tables, and use extensive flow caching, such that packets on existing flows would match precompiled actions without having to traverse tables. *Provide a fast packet classification algorithm for cases with large numbers of rules and tables.*

While solving Goal #5 dramatically improved performance for existing flows (e.g. all TCP packets following a SYN), we found a few applications pushing many thousands of rules into their flow tables (for example, a distributed router BGP peering with customers, using VFP as its FIB), which slowed down our flow compiler. We needed to design an efficient packet classifier to handle performance for these cases.

6. *Implement an efficient mechanism to offload flow policy to programmable NICs, without assuming complex rule processing.*

As we scaled to 40G+ NICs, we wanted to offload policy to NICs themselves to support SR-IOV [22, 23] and let NICs indicate packets directly to VMs while applying relevant VFP policy. However, as controllers created more flow tables with more rules, we concluded that directly offloading those tables would require prohibitively expensive hardware resources (e.g. large TCAMs, matching in series) for server-class NICs. So instead of trying to offload classification operations, we wanted an offload model that would work well with our precompiled exact-match flows, requiring hardware to only support accessing a large table of cached flows in DRAM, and support for our associated action language.

2.3 Non-Goals

The following are goals we've seen in other projects, which based on our experiences we chose not to pursue:

1. *Providing cross-platform portability.*

Portability is difficult to achieve with a high performance datapath in the kernel. Projects such as OVS have done this by splitting into a kernel fastpath and a portable userspace slowpath with policy – but this comes at a cost of over an order of magnitude slowdown when packets take the slowpath [16]. We run entirely on one host OS, so this wasn't a goal for us.

2. *Supporting a network / remote configuration protocol bundled with VFP itself.*

OpenFlow contains both a network programming model, as well as a wire protocol for configuring rules over the network. The same is true of OVS and the OVSDB protocol. In order to enable different controller models of managing policy (e.g. a rule push model, or the VL2 Directory System pull model), we instead decoupled VFP as a vswitch from the agents that implement network configuration protocols, and focused on providing a high performance host API.

3. *Providing a mechanism to detect or prevent controllers from programming conflicting policy*

Much literature [17-21] describes attempts to detect or prevent conflicts or incorrect policy in flow table or rule matching systems. Despite our first goal of supporting multiple controllers programming VFP in parallel without interfering with each other, we concluded early on that explicit conflict management was neither a feasible nor necessary goal, for several reasons. Programming VFP on behalf of a VM is a protected operation that only our controllers can perform, so we are not worried about malicious controllers. In addition, we concluded it was impossible to tell the difference between a misprogrammed flow table overwriting another flow table's actions by accident, and a flow table designed to filter the output of another table. Instead we focused on tooling to help developers validate their policy and interactions with other policies.

3. Overview and Comparison

As a motivating example throughout the paper, we consider a simple scenario requiring 4 host policies used for O(1M) VMs in a cloud. Each policy is programmed by its own SDN controller and requires both high performance and SR-IOV offload support: A VL2-style VNET, an Ananta-style load balancer, a stateful firewall, and per-destination traffic metering for billing purposes. We begin by evaluating this against existing solutions to demonstrate the need for a different approach, which we describe. Sections 4-7 then detail VFP's core design.

3.1 Existing solutions: Open vSwitch

While Linux and Windows support bridging [26-28] between multiple interfaces, which can be used as a vswitch, these bridges don't apply SDN policy. Other public clouds such as Google have described [25] using host SDN policy, but details are not public. OVS is the primary solution today to provide vswitch-based SDN, and so (as of version 2.5) is our main comparison point.

We believe OVS has had a great positive impact in making programmable host networking widely available. Many OVS design choices were driven by OVS-specific goals such as cross-platform support and the requirements of shipping in the Linux kernel[1] [1]. Combined with OVS's use of OpenFlow, these designs enable deployments with controllers managing virtual switches and physical switches via the same protocols, which was a non-goal for our host-based networking model. OVS also supports many protocols useful for physical switches such as STP, SPBM, BFD, and

[1] Windows is always backwards compatible with drivers, so we can ship a single driver compatible with all recent Windows versions without needing kernel integration.

IGMP Snooping [3], that we don't use.

Partially as a result of OpenFlow in particular, however, aspects of OVS make it unsuitable for our workload:

- OVS doesn't natively support true independent multi-controller models, as is required when our VL2 and Ananta applications are controlled separately. The underlying OpenFlow table model is unsuitable for multi-controller use cases – table rules specify explicit GOTOs to next tables, causing controllers to tie their policy together. Also, tables can only be traversed in the forward direction, whereas multi-controller scenarios require packets to traverse tables in the reverse direction for outbound packets as for inbound, so that packets will be in a consistent state when matching that controller's policy in either direction. VFP solves this with explicit table layering (§5.2).
- OVS doesn't natively support stateful actions like NAT in its MAT model, required by our Ananta example (our firewall is stateful too) – in both cases controllers need to operate on connections as a base primitive rather than packets. OpenFlow provides only for a packet model, however. OVS recently added support for sending packets to the Linux connection tracker to enable stateful firewall, but it's not exposed as a MAT and doesn't easily support a NAT, which requires explicit bidirectional stateful tables so that the NAT is reversed on a flow's return path. VFP solves this with stateful layers (§5.2).
- OVS's VTEP Schema requires explicit tunnel interfaces to implement VL2-style VNETs rather than allowing the controller to specify its own encap / decap actions, which aren't natively supported in OpenFlow[2]. This hardcodes a model of a VNET in the dataplane rather than allowing the controller to define how the VNET works (Goal 3). Adding complex VNET logic like ECMP routing can be difficult in this schema and requires vswitch changes, rather than policy changes. VFP supports all of these directly in its MAT (§5.3) by modeling encap/decap as actions.
- OVS doesn't support a VL2-style Directory System, required to dynamically look up Customer Address to Physical Address mappings. OpenFlow's design lacks the scalability to support large VNETs this way – OpenFlow exception packets must all go back to the central controller, and in OVS, VTEPs on all hosts are expected to be updated any time a mapping changes. This is OK for NSX/vSphere, which support up to 1000 hosts [30], but we found this unusable at our scale. VFP solves this by combining the schema-free MAT model with efficient asynchronous I/O exception requests (§5.5.1) that an agent can redirect to services separate from the controller.
- OVS doesn't have a generic offload action language or API that can support combinations of policy such as an Ananta NAT plus a VL2 encap. While SR-IOV offloads have been implemented on top of OVS builds by NIC vendors for specific workloads (such as VTEP schema) [31], doing general purpose offloads requires hardware to support the complex multi-table lookups of the original policy (e.g. [32]) that we've found quite costly in practice. VFP's Header Transposition language (§6.1.2, 9.3) enables SR-IOV support for all policy with only a single table lookup in hardware.

Thus we need a different design for our policy.

[2] While OpenFlow can support header pushes like MPLS tags as an action, it doesn't work for VNETs, e.g. VXLAN.

3.2 VFP Design

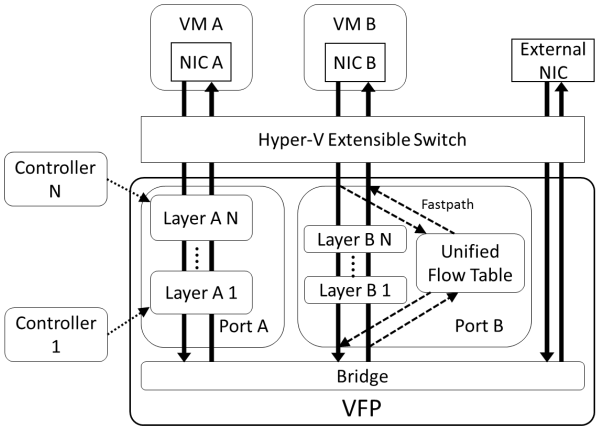


Figure 1. Overview of VFP Design

Figure 1 shows a model of the VFP design, which is described in subsequent sections. VFP operates on top of Hyper-V's extensible switch, as described in Section 4's filtering model. VFP implements MATs as layers that support a multi-controller model, with a programming model presented in Section 5. Section 6 describes VFP's packet processor, including a fastpath through unified flow tables and a classifier used to match rules in the MAT layers. Section 7 presents the switching model of VFP's bridge.

4. Filtering Model

VFP filters packets in the OS through MAT flow table policy. The filtering model is described below.

4.1 Ports and NICs

The core VFP model assumes a switch with multiple ports which are connected to virtual NICs (VNICs). VFP filters traffic from a VNIC to the switch, and from the switch to a VNIC. All VFP policy is attached to a specific port. From the perspective of a VM with a VNIC attached to a port, ingress traffic to the switch is

considered to be "outbound" traffic from the VM, and egress traffic from the switch is considered to be "inbound" traffic to the VM. VFPAPI and its policies are based on the inbound/outbound model.

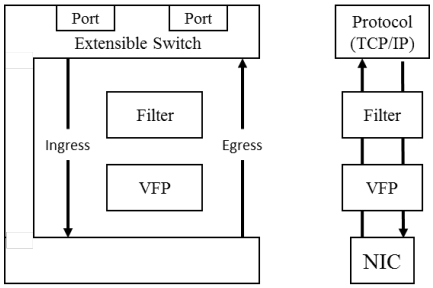


Figure 2. Hyper-V Switch Extensibility vs NIC Filtering

VFP implements a switch abstraction interface to abstract out different environments, instantiations of which provide logic for management (e.g. create / delete / connect / disconnect) of ports, VNICs, and associated objects. This interface supports both a Hyper-V switch and a filter for native hosts, shown in Figure 2.

4.2 Hyper-V Switch Extensibility

Hyper-V includes a basic vswitch [28] to bridge VNICs to a physical NIC. The switch is extensible, allowing filters to plug in and filter traffic to and from VNICs.

VFP acts as a Forwarding Extension to Hyper-V's vswitch – it simply replaces the entire switch logic with itself. Using this model allows us to keep our policy module and virtual switching logic (VFP) separate from the Hyper-V infrastructure to deliver packets to and from VMs, improving modularity and serviceability.

VFP in this mode supports PacketDirect [11], which allows a client to poll a NIC with very low overhead.

5. Programming Model

VFP's core programming model is based on a hierarchy of VFP objects that controllers can create and program to specify their SDN policy. The objects are:

- *Ports*, the basic unit that VFP policy filters on.
- *Layers*, the stateful flow tables that hold MAT policy.
- *Groups*, entities to manage and control related groups of rules within a layer.
- *Rules*, the match action table entries themselves.

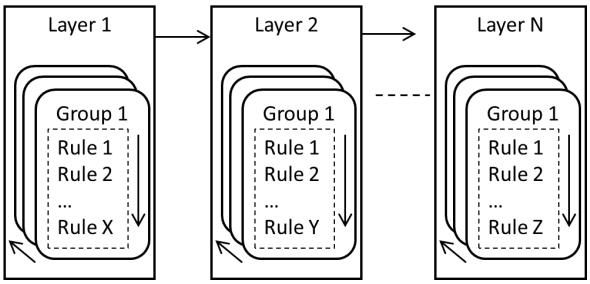


Figure 3. VFP Objects: Layers, Groups, and Rules

5.1 Ports

VFP's policy is implemented on a per-port basis – each port has match action tables which can sit on the inbound or outbound path of the port, acting as filters. Since our controllers generally want to program policy on behalf of a VM or VNIC, this clean separation of ports allows controllers to independently manage policy on different VMs, and instantiate and manage flow tables only on ports where they are needed – for example a VM in a virtual network may have tables to encapsulate and decapsulate traffic into tunnels, which another VM not in a virtual network wouldn't need (the VNET controller may not even be aware of the other VM, which it doesn't need to manage).

Policy objects on VFP are arranged in fixed object hierarchies, used to specify which object a given API call is referencing, such as Layer/Group/Rule. All objects are be programmed with a priority value, in which order they will be processed by rule matching.

5.2 Layers

VFP divides a port's policy into layers. Layers are the basic Match Action Tables that controllers use to specify their policy. They can be created and managed separately by different controllers, or one controller can create several layers. Each layer contains inbound and outbound rules and policies that can filter and modify packets. Logically, packets go through each layer one by one, matching rules in each based on the state of the packet after the action performed in the previous layer. Controllers can specify the ordering of their layers in a port's pipeline with respect to other layers, and create and destroy layers dynamically during operation.

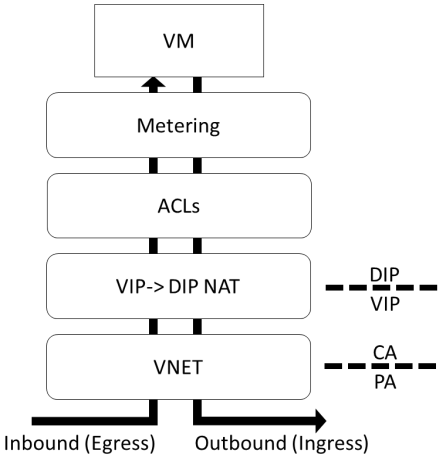


Figure 4. Example VFP Layers with Boundaries

Critically, packets traverse layers in the opposite order when inbound than when outbound. This gives them a "layering" effect when controllers implement opposite policy on either side of a layer. Take for example a load balancing layer implementing the Ananta NAT design.

On the inbound direction, the layer NATs connections destined to a Virtual IP (a VIP) to a Direct IP (DIP) behind the VIP – in this case the IP of the VM. On the outbound direction, it NATs packets back from DIP to VIP. The layer thus implements an address space boundary – all packets above it are in "DIP Space", and all packets below it are in "VIP Space". Other controllers can choose to create layers above or below this NAT layer, and can plumb rules to match VIPs or DIPs respectively – all without coordination with or involvement of the NAT controller.

Figure 4 shows layers for our SDN deployment example. VL2 is implemented by a VNET layer programmed by a virtual network controller, using tunneling for Customer Addresses (CAs) so that packets can traverse a physical network in Physical Address (PA) space recognized by physical switches in the path between VMs. This layer creates a CA / PA boundary by having encapsulation rules on the outbound path and decapsulation rules in the inbound path. In addition, an ACL layer for a stateful firewall sits above our Ananta NAT layer. The security controller, having placed it here with respect to those boundaries, knows that it can program policies matching DIPs of VMs, in CA space. Finally a metering layer used for billing sits at the top next to the VM, where it can meter traffic exactly as the customer in the VM sees it – all traffic that made it in and all traffic that was sent out from the VM.

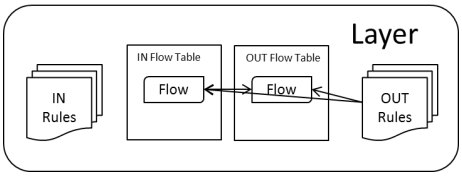


Figure 5. A Layer with a stateful flow

Layering also gives us a good model on which to implement stateful policy. Since packets on a given connection should be in the same IP/Port state on both the inbound and outbound path, we can keep flow state on a layer by assuming that a TCP or UDP 5-tuple (SrcIP, DstIP, IP Protocol, SrcPort, DstPort) will be the opposite on each side of the layer, and encoding that in a hash table of all connections in either direction. When a stateful rule is matched, it creates both an inbound and outbound flow in the layer flow tables, with the flow in the direction of the rule having the action of the rule, and the opposite direction taking the opposite action, to maintain layering. These inbound and outbound flows are considered *paired* – their actions simply change the packet to the state of the opposite flow in the pair rather than carrying their own action context.

When processing packets, VFP searches for a single rule in each layer to match by searching the groups of rules inside a layer for a matching rule. That rule's action is then performed on the packet – only one rule can match a given packet in a given layer (other matching rules of lower priority are ignored).

5.3 Rules

Rules are the entities that perform actions on matching packets in the MAT model. Per Goal #3, rules allow the controller to be as expressive as possible while minimizing fixed policy in the dataplane. Rules are made up of two parts: a condition list, specified via a list of conditions, and an action, both described below.

5.3.1 Conditions

When a VFPAPI client programs a rule, it provides a descriptor with a list of conditions. Conditions have a type (such as source IP address), and a list of matching values (each value may be a singleton, range, or prefix). For a condition to match a packet, any of the matching values can match (an OR clause). For a rule to match, all conditions in the rule much match (an AND clause).

5.3.2 Actions

A rule descriptor also has an action. The action contains a type and a data structure specific to that type with data needed to perform the rule (for example, an encapsulation rule takes as input data the source / destination IP addresses, source / destination MACs, encapsulation format and key to use in encapsulating the packet). The action interface is extensible - example conditions and actions are listed in Figure 6.

Rules are implemented via a simple callback interface (Initialize, Process Packet, Deinitialize) so as to make the base VFP platform easily extensible. If a rule type supports stateful instantiation, the process handler will create a pair of flows in the layer as well – flows are also typed and have a similar callback interface to rules. A stateful rule includes a flow time to live, which is the time that flows it creates will remain in the flow table after the last packet matches (unless expired explicitly by the TCP state machine described in §6.4.2).

Conditions	Actions
Source/Dest MAC	Allow/Block (Stateful/Stateless)
Source/Dest IP	
Source/Dest TCP Port	NAT (L3/L4), (Stateful/Stateless)
Source/Dest UDP Port	
GRE Key	Encap/Decap
VXLAN VNI	QoS – Rate Limit, Mark DSCP, Meter
VLAN ID	Encrypt/Decrypt
Metadata From Previous Layer	Stateful Tunneling
	Routing (ECMP)

Figure 6. Example Conditions and Actions

5.3.3 User Defined Actions

In addition to a large set of actions we'd created over

time, in VFPv2 we added user-defined actions to further Goal #3 – allowing the controllers to create their own rule types using a language for header field manipulations (Header Transpositions, see §6.1.2). This allows extending the base VFP action set without writing code to implement an action in the datapath.

5.4 Groups

Rules on a layer are organized into logical groups for management purposes. Groups are the atomic unit of policy in VFP – clients can transactionally update them. When classifying packets, VFP iterates through groups in a layer to find the highest priority rule in each group that matches the packet. By default, VFP will select the rule matched by the last group in the list. A rule can be marked "terminating," meaning that if it ever matches it will be applied immediately without traversing further groups. Groups can have conditions just like rules – if a group's condition doesn't match, VFP will skip it.

Below are two examples of how we've seen groups used for management of different policies in one layer:

- For VMs with Docker-style containers [35], each with its own IP, groups can be created and managed on a per-container basis by setting an IP condition on them.
- For our stateful firewall, infrastructure ACLs and customer ACLs can be expressed as two groups in a layer. Block rules would be marked terminating – if either group blocks it, a packet is dropped. Only if both sets of rules allowed a packet does it go through.

In addition to priority-based matching, individual groups can be Longest Prefix Matching on a condition type (for example, destination IP) to support routing scenarios. This is implemented as a compressed trie.

5.5 Resources

MATs are a good model for programming general network policy, but on their own aren't optimal for every scenario, especially ones with exception events. VNET requires a CA->PA lookup on outbound traffic (using a Directory System). Rules alone aren't optimal for such large mapping tables. So we support an extensible model of generic resources – in this case, a hash table of mappings. A resource is a port-wide structure that any rule on a port can reference. Another example is a range list, which can implement a dynamic source NAT rule of the form described in Ananta.

5.5.1 Event Handling / Lookups

Fast eventing APIs are required for many SDN applications where there is a lookup miss. We generally handle events in the context of resources – e.g. if an encap rule looks up a PA/CA mapping resource and misses, a VFPAPI client can register an efficient callback mechanism using async I/O and events. We use the same mechanism for Ananta NAT port exhaustion.

6. Packet Processor and Flow Compiler

As we scaled production deployments of VFP, and SDN became more widely used, it became necessary to write a new VFP datapath for improved performance and scalability across many rules and layers. Our work to improve performance, without losing the flexibility and programmability of VFPAPI, is described below.

6.1 Metadata Pipeline Model

VFP's original 2012 release, while performant under the workloads it was designed for, didn't scale well when the host SDN model took off even faster than we expected and many new layers were created by controllers. VFP rules and flows were implemented as callbacks which took a packet as input and modified its buffer - the next layer would have to reparse it. The original rule classification logic was linear match (with stateful flows accelerating this). At 10+ layers with thousands of rules, we needed something better.

A primary innovation in VFPv2 was the introduction of a central packet processor. We took inspiration from a common design in network ASIC pipelines e.g. [34] – parse the relevant metadata from the packet and act on the metadata rather than on the packet, only touching the packet at the end of the pipeline once all decisions have been made. We compile and store flows as we see packets. Our just-in-time flow compiler includes a parser, an action language, an engine for manipulating parsed metadata and actions, and a flow cache.

6.1.1 Unified FlowIDs

VFP's packet processor begins with parsing. The relevant fields to parse are all those which can be matched in conditions (from §5.3.1). One each of an L2/L3/L4 header (as defined in table 1) form a header group, and the relevant fields of a header group form a single FlowID. The tuple of all FlowIDs in a packet is a Unified FlowID (UFID) – the output of the parser.

6.1.2 Header Transpositions

Our action primitives, Header Transpositions (HTs), so called because they change or shift fields throughout a packet, are a list of paramaterizable header actions, one for each header. Actions (defined in table 2) are to *Push* a header (add it to the header stack), *Modify* a header (change fields within a given header), *Pop* a header (remove it from the header stack), or *Ignore* a header (pass over it). HTs are parameterized with all fields in a given header that can be matched (so as to create a complete language – any valid VFP flow can be transformed into any other valid VFP flow via exactly one HT). Actions in a HT are grouped into header groups. Table 3 shows examples of a NAT HT used by Ananta, and encap/decap HTs used by VL2.

As part of VFPv2, all rule processing handlers were

updated to take as input a FlowID and output a transposition. This has made it easy to extend VFP with new rules, since implementing a rule doesn't require touching packets – it's a pure metadata operation.

Table 1. Valid Parameters for Each Header Type

Header	Parameters
Ethernet (L2)	Source MAC, Dest MAC
IP (L3)	Source IP, Dest IP, ToS (DSCP+ECN)
Encapsulation (L4)	Encapsulation Type, Tenant ID, Entropy (Optional)
TCP/UDP (L4)	Source Port, Dest Port, TCP Flags (note: does not support Push/Pop)

Table 2. Header Transposition Actions

Action	Notes
Pop	Remove this header.
Push	Push this header onto the packet. All header parameters for creating the new header are specified.
Modify	Modify this header. All header parameters needed are optional, but at least one is specified.
Ignore	Leave this header as is.

Table 3. Example Header Transpositions

Header	NAT	Encap	Decap	Encap+ NAT
Outer Ethernet	Ignore	Push (SMAC, DMAC)	Pop	Push (SMAC, DMAC)
Outer IP	Modify (SIP,DIP)	Push (SIP,DIP)	Pop	Push (SIP,DIP)
GRE	Not Present	Push (Key)	Pop	Push (Key)
Inner Ethernet	Not Present	Modify (DMAC)	Ignore	Modify (DMAC)
Inner IP	Not Present	Ignore	Ignore	Modify (SIP,DIP)
TCP/UDP	Modify (SPt,DPt)	Ignore	Ignore	Modify (SPt,DPt)

6.1.3 Transposition Engine

VFP creates an action for a UFID match by composing HTs from matched rules in each layer, as in Pseudocode 1. For example, a packet passing the example Ananta NAT layer and the VL2 VNET encap layer may end up with the composite Encap+NAT transposition in Table 3. This transposition engine also contains logic to apply a transposition to an actual packet, by breaking the final transposition down into a series of steps (NAT, encap, decap) that can be applied by a packet modifier.

6.1.4 Unified Flow Tables and Caching

The intuition behind our flow compiler is that the action for a UFID is relatively stable over the lifetime of a flow – so we can cache the UFID with the resulting HT from the engine. Applications like Ananta create per-connection state already, so it's not expensive to cache this whole unified flow (UF) per TCP/UDP flow. The resulting flow table where the compiler caches UFs is called the Unified Flow Table (UFT).

```
Process(UFID input, Port port):
  Transposition action = {0};
  For each layer in port.layers:
    UFID localId = Transpose(input, action);
    Rule rule = Classify(layer, localId);
    action = action.compose(rule.process(localId));
  return composite;
```

Pseudocode 1. Transposition Engine

With the UFT, we segment our datapath into a fastpath and a slowpath. On the first packet of a TCP flow, we take a slowpath, running the transposition engine and matching at each layer against rules. On subsequent packets, VFP takes a fastpath, matching a unified flow via UFID, and applying a transposition directly. This operation is independent of the layers or rules in VFP.

The UFT is used similarly to the OVS microflow cache to skip tables, and scales well across CPUs because it requires no write lock to match. However, a key difference for our workload is the HT, which combines encap/decap with header modification. This allows us to have a single flow for all actions rather than one before and after a tunnel interface, and is critical for offloading flows with only a single hardware table (§9.3).

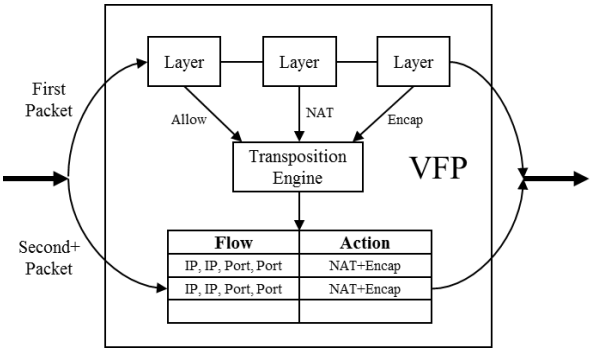


Figure 7. VFP Unified Flow Table

6.2 Action Contexts

Some rule actions have side effects beyond header modification, or take action on packet payloads. Examples include metering to a global counter (supporting our example metering layer), or encrypting packet payloads. For these actions, HTs can be extended with *Action Contexts* which can implement arbitrary logic via callback. An Action Context can be added to an HT (and the resulting UF) by a rule. This allows rules to extend the packet actions themselves even though they are not matched for every packet.

6.3 Flow Reconciliation

A requirement of the VFP flow compiler is transparency to VFPAPI clients. This means that if a controller changes the rules in a layer, the new rules should be

applied to subsequent packets even if a UF exists. This is supported by a reconciliation engine in VFP.

The reconciliation engine maintains a global generation number on each port. When a UF is created, it's tagged with the current generation number at creation time. Upon policy update, the port generation is incremented.

VFP implements lazy reconciliation, reconciling a UF only when matching a UF whose generation number is less than the port's current generation number. The UF is then *simulated* against the current rules on the port, by running its UFID through the transposition engine, and determining if the resulting HT has changed.

6.4 Flow State Tracking

By default, the expiration policy for UFs is to expire them after some configurable period of time. However, this is not efficient for short flows and leads to large numbers of UFs idling in the UFT. Instead, for TCP flows we can expire them by tracking the state of the underlying connections. This requires determining which UF should pair with a UF in the opposite direction to form a bidirectional connection.

6.4.1 Flow Pairing

Unlike layer flows, we cannot pair UFs just by reversing the FlowID – UF pairs can be asymmetric, for example if a connection is tunneled to the VM on inbound but returned directly without tunneling on the outbound side (e.g. the Direct Server Return feature in Ananta). The key idea in our solution is pairing connections on the VM-side of the VFP pipeline rather than the network side (e.g. the UFID of a connection on the inbound path after processing should be the opposite of the UFID on the outbound path before processing).

When an inbound UF is created by an inbound packet, we create an outbound UF to pair it to by reversing the UFID of the packet after the inbound action, and simulating it through the outbound path of that port, generating a full UF pair for the connection. For a new outbound UF, we wait for an inbound packet to try to create an inbound UF – when the new UF looks up the reverse UFID it will find an existing flow and pair itself.

6.4.2 TCP State Tracking

Once we have established a pairing of UFs, we use a simple TCP state machine in VFP to track them as connections. For example, new flows are created in a probationary half-open state – only when a three-way handshake is verified with proper sequence numbers does it become a full flow (this helps protect against SYN floods). We can also use this state machine to track FIN handshakes and RSTs, to expire flows early. We also track connections in TIME_WAIT, allowing NAT rules to determine when they can reuse ports safely. VFP tracks port-wide statistics such as average RTT, retransmits, and ECN marks, which can be useful in diagnosing VMs with networking issues [37].

6.5 Packet Classification

In practice, VFPv2's UFT datapath solved performance and scalability issues for most packets (even short lived flows are usually at least 10 packets). However, rule matching is still a factor for scenarios with thousands of rules, such as complex ACLs or routes. We implement a better classification algorithm for these cases.

We support 4 classifier types: a compressed trie, an interval tree, a hash table, and a list. Each classifier can be instantiated on each condition type from §5.3.1. We assign each rule to a classifier based on a heuristic described in detail in [40] to optimize total matching time in a VFP group. In practice we have found this to be successful at handling a wide range of rule types that users plumb, such as 5-tuple ACLs with IP ranges, and large routing tables.

7. Switching Model

In addition to SDN filtering, VFP also forwards traffic to ports. VFP's forwarding plane is described below.

7.1 Packet Forwarding

VFP implements a simple bridge, forwarding packets by destination MAC address to the port attached to the VNIC created with that MAC. This can be an outer MAC, or an inner MAC inside a VXLAN/NVGRE encaped packet for VMs running in a virtualized mode. Efficient spreading based on this policy is widely supported by NICs as Virtual Machine Queue [33].

7.2 Hairpinning and Mirroring

Gateway VMs are often used to bridge different tunnels or address spaces in different routing domains. To accelerate these workloads, we support hairpin rules in VFP layers, which can redirect packets to a VNIC back out through VFP ingress processing, either on the same or a different VNIC. This enables high speed gateways without the overhead of sending packets to a VM. This policy also supports programmable port mirroring.

7.3 QoS

VFP supports applying max cap policies on transmit and receive to ports or groups of ports on a switch. This is implemented using atomic operations to update token bucket counters and interlocked packet queues for high performance. Bandwidth reservations across ports are also supported. Our algorithm for measurement-based weighted fair sharing is described in [39].

8. Operational Considerations

As a production cloud service, VFP's design must take into account serviceability, monitoring and diagnostics.

8.1 Rebootless Updates

In an update, we first pause the datapath, then detach

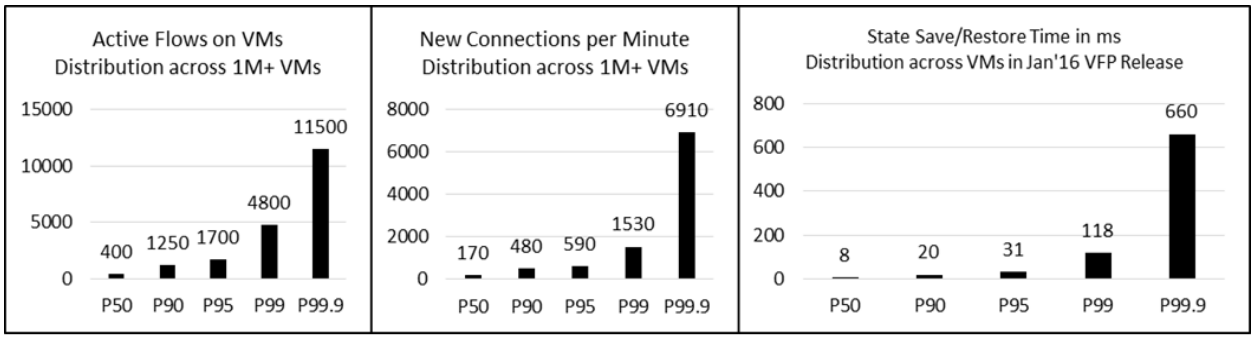


Figure 8. Example VFP Production Monitoring Statistics

VFP from the stack, uninstall VFP (which acts as a loadable kernel driver), install a new VFP, attach it to the stack, and restart the datapath. This operation typically completes in <1s, and looks like a brief connectivity blip to VMs, while the NIC stays up.

8.2 State Save/Restore

Because we support stateful policy such as ACLs and NAT, a rebootless VFP update by default forces VMs to reset all TCP connections, since flow state is lost. As we saw more and more frequent updates, we concluded we needed to build State Save/Restore (SSR) functions to eliminate impact of VFP updates to VMs.

We support serialization and deserialization for all policy and state in VFP on a port. Every VFP object has a serialization and deserialization handler, including layers/groups/rules/flows, rule contexts, action contexts, UFs, HTs, resources and resource entries, and more. All objects are versioned, so if structures are updated, SSR can support multiple source object versions.

8.2.1 VM Live Migration

VFP also supports live migration of VMs. In this case, the port state is serialized out of the original host and deserialized on the new host during the VM blackout time of the migration. VFP policies/rules are updated on the new host by all VFPAPI clients based on policy that may have changed in migration, such as the VM physical address. VFP flow reconciliation (§6.3) then handles updating flows, keeping TCP connections alive.

8.3 Monitoring

VFP implements over 300 performance counters and flow statistics, on per port, per layer, and per rule bases, as well as extensive flow statistics. This information is continuously uploaded to a central monitoring service, providing dashboards on which we can monitor flow utilization, drops, connection resets, and more, on a VM or aggregated on a cluster/node/VNET basis. Figure 8 shows distributions measured in production for the number of active flows and rate of new connections per VM, and SSR time for a recent VFP update.

8.4 Diagnostics

VFP provides diagnostics for production debugging, both of VFP itself and for VFPAPI clients. The transposition engine's simulation path can be queried on arbitrary UFIDs to provide a trace of how that UFID would behave across all rules/flows in VFP. This enables remotely debugging incorrect rules and policies.

VFP tracing, when enabled, provides detailed logs of actions performed on the data path and control path.

If we need to track a bug in VFP's logic, SSR can snapshot a port's state in production, and restore it on a local test machine, where packets can then be simulated using the above diagnostic under a kernel debugger.

9. Hardware Offloads and Performance

As we continue to scale up our cloud servers, we are very performance and cost sensitive. VFP implements several hardware offloads, described below.

9.1 Stateless Tunneling Offloads

Commercial NICs have been available since 2013 with the ability to parse NVGRE/VXLAN, and support stateless checksum, segmentation, and receive scaling offloads with them. VFP programs NICs with these offloads for tunneling, which are widely deployed in our datacenter. This has largely eliminated the performance overhead of encap/decap to our platform – with offloads we can support encap at 40Gbps line rate on our NICs.

9.2 QoS Offloads

Many commercial NICs support transmit max caps, and bandwdith reservations, across transmit queues. We have implemented and deployed an interface to offload port-level QoS policy from §7.3. This removes the overhead of applying QoS through software.

9.3 Offloading VFP Policy

Many attempts to offload SDN policy and accomplish Goal #7 of SR-IOV support focus on offloading packet classification, such that no packets traverse the host. We've found this impractical on server NICs, as it often requires large TCAMs, CPU cores or other specialized hardware when doing lookups on 10+ tables in series.

However, offloading Unified Flows as defined in §6 turns out to be much more tractable. These are exact match flows representing each connection on the system, and so they can be implemented via a large hash

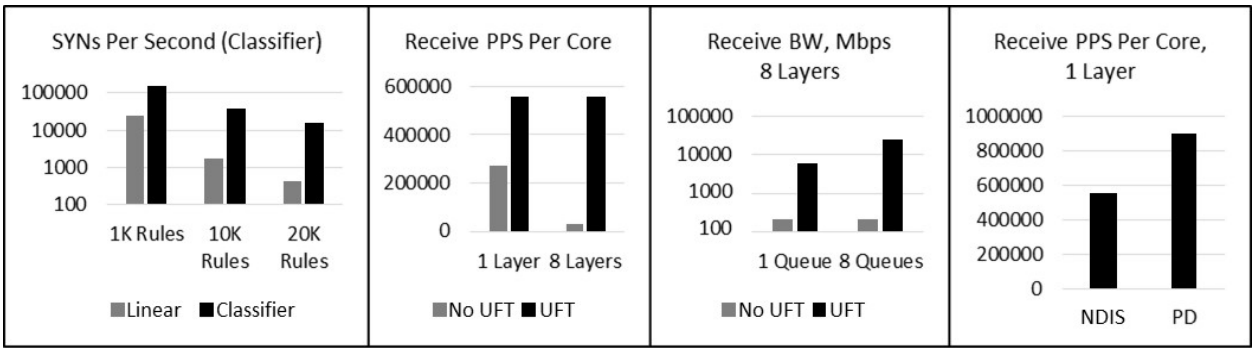


Figure 9. Selected VFP Performance Data, 2.9Ghz Xeon Sandy Bridge CPU

table, typically in inexpensive DRAM. In this model, the first packet of a new flow goes through software classification to determine the UF, which is then offloaded so that packets take a hardware path.

We've used this mechanism to enable SR-IOV in our datacenters with VFP policy offload on custom hardware we've deployed on all new Azure servers. Our VMs reach 25Gbps+ VNICs at line rate with near-zero host CPU and <25µs e2e TCP latencies inside a VNET.

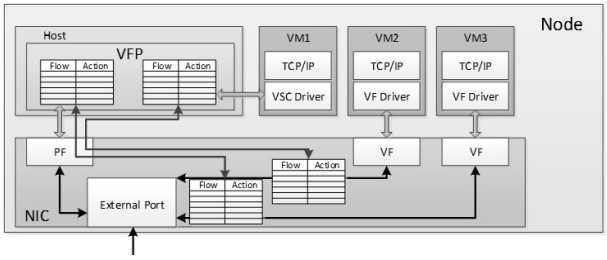


Figure 10. VFP Offloading Flow Tables per Port

9.4 Performance

In Figure 9, we first see that under a TCP SYN load the classification algorithm in §6.5 improves by 1-2 orders of magnitude over linear match a distribution of random rules. Measuring VFP's fastpath against long lived flows, we next see that on layers of 200 rules each, UFT caching improves performance even at one layer, but more dramatically as more layers are added. We then see that due to locking, in VFPv1 performance doesn't improve as we scale to 8 CPUs, while UFT scales well. Finally, we see that received packets per second improves by over 60% with PacketDirect.

10. Experiences

We have deployed 21 major releases of VFP since 2012. VFP runs on all Azure servers, powering millions of VMs, petabits per second of traffic, and providing load balancing for exabytes of storage, in hundreds of datacenters in over 30 regions across the world. In addition, we are releasing VFP publicly as part of Windows Server 2016 for on-premises workloads.

10.1 Results

We believe we accomplished all of our goals from §2:

1. We have had multiple independent controllers successfully program VFP for several years. New controllers have deployed SDN applications by inserting layers without changes in other controllers.
2. Every single connection in our datacenters is treated statefully by VFP – all pass through stateful ACLs, and many pass through a stateful NAT.
3. The definition of a VNET, a load balancer, and other policies have changed over time as newer, richer semantics were implemented. Most of these were policy changes without any change to VFP.
4. We've pushed dozens of rebootless updates to VFP.
5. The introduction of UFT dramatically improved VFP performance, especially for scenarios with large numbers of layers. This has helped us scale our servers to 40G+ NICs, with many more VMs.
6. The VFP packet classification algorithm has sped up classification on real production workloads by 1-2 orders of magnitude over linear search.
7. We have successfully offloaded VFP flows to flow-programmable hardware and deployed SR-IOV.

10.2 Lessons Learned

Over 5 years of developing and supporting VFP, we learned a number of other lessons of value:

- **L4 flow caching is sufficient.** We didn't find a use for multi-tiered flow caching such as OVS megaflows. The two main reasons: being entirely in the kernel allowed us to have a faster slowpath, and our use of a stateful NAT created an action for every L4 flow and so reduced the usefulness of ternary flow caching.
- **Design for statefulness from day 1.** The above point is an example of a larger lesson: support for stateful connections as a first class primitive in a MAT is fundamental and must be considered in every aspect of a MAT design. It should not be bolted on later.
- **Layering is critical.** Some of our policy could be implemented as a special case of OpenFlow tables with custom GOTOs chaining them together, with separate inbound and outbound tables. We found,

however, that our controllers needed clear layering semantics or else they couldn't reverse their policy correctly with respect to other controllers.

- **GOTO considered harmful.** Controller developers will implement policy in the simplest way needed to solve a problem, but that may not be compatible with future controllers adding policy. We needed to be vigilant in not only providing layering, but enforcing it to prevent this. We see this layering enforcement not as a limitation compared to OpenFlow's GOTO table model, but instead as the key feature that made multi-controller designs work for 4 years running.
- **IaaS cannot handle downtime.** We found that customer IaaS workloads cared deeply about uptime for each VM, not just their service as a whole. We needed to design all updates to minimize downtime, and provide guarantees for low blackout times.
- **Design for serviceability.** SSR (§8.2) is another design point that turned out to pervade all of our logic – in order to regularly update VFP without impact to VMs, we needed to consider serviceability in any new VFP feature or action type.
- **Decouple the wire protocol from the data plane.** We've seen enough controllers/agents implement wire protocols with different distributed systems models to support O(1M) scale that we believe our decision to separate VFPAPI from any wire protocol was a critical choice for VFP's success. For example, bandwidth metering rules are pushed by a controller, but VNET required a VL2-style directory system (and an agent that understands that policy comes from a different controller than pulled mappings) to scale.

  Architecturally, we believe it helps to view the resulting "Smart Agents" as part of a distributed controller application, rather than part of the dataplane, and we consider VFP's OS level API the real common Southbound API [36] in our SDN stack, where different applications meet.
- **Conflict resolution was not needed.** We believe our choice in §2.3 to not focus on automated conflict resolution between controllers was correct, as this was never really an issue when we enforced clean layering between the controllers. Good diagnostics to help controller developers understand their policy's impact were more important.
- **Everything is an action.** Modeling VL2-style encap/decap as actions rather than tunnel interfaces was a good choice. It enabled a single table lookup for all packets – no traversing a tunnel interface with tables before and after. The resulting HT language combining encap/decap with header modification enabled single-table hardware offload.
- **MTU is not a major issue.** There were initial concerns that using actions instead of tunnel interfaces would cause issues with MTU. We found this not to be a real issue – we support either making use of larger MTU on the physical network to support encapsulation (this was our choice for Azure), or using a combination of TCP Maximum Segment Size clamping and fragmentation of large non-TCP frames (for deployments without large MTU).
- **MAT Scale.** In our deployments, we typically see up to 10-20 layers, with up to hundreds of groups within a layer. We've seen up to O(50k) rules per group, (when supporting customers' distributed BGP routers). We support up to 500k simultaneous TCP connections per port (after which state tracking becomes prohibitively expensive).
- **Keep forwarding simple.** §7.1 describes our MAC-filter based forwarding plane. We considered a programmable forwarding plane based on the MAT model, however, we found no scenario for complex forwarding policy or learning. We've concluded that a programmable forwarding plane is not useful for our cloud workload, because VMs want a declarative model for creating NICs with known MAC addresses.
- **Design for E2E monitoring.** Determining network health of VMs despite not having direct access to them is a challenge. We found many uses for in-band monitoring with packet injectors and auto-responders implemented as VFP rule actions. We used these to build monitoring that traces the E2E path from the VM-host boundary. For example, we implemented Pingmesh-like [24] monitoring for VL2 VNETs.
- **Commercial NIC hardware isn't ideal for SDN.** Despite years of interest from NIC vendors about offloading SDN policy with SR-IOV, we have seen no success cases of NIC ASIC vendors supporting our policy as a traditional direct offload. Instead, large multi-core NPUs [32] are often used. We used custom FPGA-based hardware to ship SR-IOV in Azure, which we found was lower latency and more efficient.

11. Conclusions and Future Work

We introduced the Virtual Filtering Platform (VFP), our cloud scale vswitch for host SDN policy in Microsoft Azure. We discussed how our design achieved our dual goals of programmability and scalability. We discussed concerns around serviceability, monitoring, and diagnostics in production environments, and provided performance results, data, and lessons from real use.

Future areas of investigation include new hardware models of SDN, and extending VFP's offload language.

Acknowledgements

We would like to thank the developers who worked on VFP since the project's inception in 2011 – Yue Zuo, Harish Kumar Chandrappa, Praveen Balasubramanian, Vikas Bhardwaj, Somesh Chaturmohta, Milan Dasgupta, Mahmoud Elhaddad, Luis Hernandez, Nathan Hu, Alan Jowett, Hadi Katebi, Fengfen Liu, Keith Mange, Randy Miller, Claire Mitchell, Sambhrama Mundkur, Chidambaram Muthu, Gaurav Poothia, Madhan Sivakumar, Ethan Song, Khoa To, Kelvin Zou, and Qasim Zuhair, as well as PMs Yu-Shun Wang, Eric Lantz, and Gabriel Silva. We thank Alireza Dabagh, Deepak Bansal, Pankaj Garg, Changhoon Kim, Hemant Kumar, Parveen Patel, Parag Sharma, Nisheeth Srivastava, Venkat Thiruvengadam, Narasimhan Venkataramaiah, Haiyong Wang, and other architects of Azure's SDN stack who had significant influence on VFP's design, as well as Jitendra Padhye, Parveen Patel, Shachar Raindel, Monia Ghobadi, George Varghese, our shepherd David Wetherall, and the anonymous reviewers, who provided valuable feedback on earlier drafts of this paper. Finally, we thank Dave Maltz, Mark Russinovich, and Albert Greenberg for their sponsorship of and support for this project over the years, and Jitendra Padhye for convincing us to write a paper about our experiences.

References

[1] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, and P. Shelar. The Design and Implementation of Open vSwitch. In NSDI, 2015.

[2] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, S. Sengupta. VL2: A scalable and flexible data center network. In SIGCOMM, 2009.

[3] Open vSwitch – An Open Virtual Switch. http://www.openvswitch.org.

[4] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, N. Karri. Ananta: Cloud scale load balancing. In SIGCOMM, 2013.

[5] N. Mckeown, T. Anderson, H. Balakrishnan, G. M. Parulkar, L. L. Peterson, J. Rexford, S. Shenker, and J. S. Turner. OpenFlow: Enabling Innovation in Campus Networks. In SIGCOMM, 2008.

[6] M. Mahalingam et al. RFC 7348: Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. 2015.

[7] M. Sridharan et al. RFC 7637: NVGRE: Network Virtualization Using Generic Routing Encapsulation. 2015.

[8] B. Pfaff et al. RFC 7047: The Open vSwitch Database Management Protocol. 2015.

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, D. Walker. P4: Programming Protocol-Independent Packet Processors. ACM Sigcomm Computer Communications Review (CCR). Volume 44, Issue #3 (July 2014).

[10] NDIS Filter Drivers. https://msdn.microsoft.com/en-us/library/windows/hardware/ff565492(v=vs.85).aspx

[11] PacketDirect Provider Interface. https://msdn.microsoft.com/en-us/library/windows/hardware/mt627758(v=vs.85).aspx.

[12] Hyper-V Extensible Switch. https://msdn.microsoft.com/en-us/library/windows/hardware/hh582268(v=vs.85).aspx

[13] Amazon Web Services. http://aws.amazon.com.

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

[15] Google Cloud Platform. http://cloud.google.com.

[16] M. Challa. OpenVswitch Performance measurements & analysis. 2014. http://openvswitch.org/support/ovscon2014/18/1600-ovs_perf.pptx

[17] A. AuYoung, Y. Ma, S. Banerjee, J. Lee, P. Sharma, Y. Turner, C. Liang, J. C. Mogul. Democratic Resolution of Resource Conflicts Between SDN Control Programs. In CoNEXT, 2014.

[18] J. Mogul, A. AuYoung, S. Banerjee, L. Popa, J. Lee, J. Mudigonda, P. Sharma, Y. Turner. Corybantic: towards the modular composition of SDN control programs. In HotNets-XII, 2013.

[19] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-Defined Networks. In NSDI, 2013.

[20] N. Foster, M. J. Freedman, R. Harrison, J. Rexford, M. L. Meola, and D. Walker. Frenetic: A High-Level Language for OpenFlow Networks. In PRESTO, 2010.

[21] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In ICFP, 2011.

[22] Overview of Single Root I/O Virtualization (SR-IOV). https://msdn.microsoft.com/en-us/windows/hardware/drivers/network/overview-of-single-root-i-o-virtualization--sr-iov-

[23] Y. Dong, X. Yang, X. Li, J. Li, K. Tan, H. Guan. High performance network virtualization with SR-IOV. In IEEE HPCA, 2010.

[24] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In SIGCOMM, 2015.

[25] A. Vahdat. Enter the Andromeda zone - Google Cloud Platform's latest networking stack. 2014. https://cloudplatform.googleblog.com/2014/04/enter-andromeda-zone-google-cloud-platforms-latest-networking-stack.html

[26] Linux Bridge. Linux Foundation. https://wiki.linuxfoundation.org/networking/bridge

[27] Network Bridge Overview. Microsoft. https://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/hnw_understanding_bridge.mspx?mfr=true

[28] Hyper-V Virtual Switch Overview. Microsoft. https://technet.microsoft.com/en-us/library/hh831823(v=ws.11).aspx

[29] VTEP Schema. Open vSwitch. http://openvswitch.org/support/dist-docs/vtep.5.html

[30] vSphere 6.0 Configuration Maximums. VMWare. 2016. https://www.vmware.com/pdf/vsphere6/r60/vsphere-60-configuration-maximums.pdf

[31] Mellanox Presentation on OVS Offload. Mellanox. 2015. http://events.linuxfoundation.org/sites/events/files/slides/Mellanox%20OPNFV%20Presentation%20on%20OVS%20Offload%20Nov%2012th%202015.pdf

[32] Open vSwitch Offload and Acceleration with Agilio CX Intelligent Server Adapters. Netronome. https://www.netronome.com/media/redactor_files/WP_OVS_Benchmarking.pdf

[33] Virtual Machine Queue. Microsoft. https://msdn.microsoft.com/en-us/windows/hardware/drivers/network/virtual-machine-queue--vmq-

[34] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, N. McKeown. Implementing an OpenFlow switch on the NetFPGA platform. In ANCS, 2008.

[35] D. Merkel. Docker: lightweight Linux containers for consistent development and deployment. Linux Journal, Issue 239, March 2014.

[36] C. Beckmann. Southbound SDN API's. In IETF 84 SDNRG Proceedings, 2012. https://www.ietf.org/proceedings/84/slides/slides-84-sdnrg-7.pdf

[37] M. Moshref, M. Yu, R. Govindan, A. Vahdat. Trumpet: Timely and Precise Triggers in Data Centers. In SIGCOMM, 2016.

[38] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, J. Rexford. PISCES: A Programmable, Protocol-Independent Software Switch. In SIGCOMM, 2016.

[39] K. To, D. Firestone, G. Varghese, J. Padhye. Measurement Based Fair Queuing for Allocating Bandwidth to Virtual Machines. In HotMiddlebox, 2016.

[40] D. Firestone, H. Katebi, G. Varghese. Virtual Switch Packet Classification. In Microsoft TechReport, MSR-TR-2016-66, 2016.

SCL: Simplifying Distributed SDN Control Planes

Aurojit Panda[†] Wenting Zheng[†] Xiaohe Hu[‡] Arvind Krishnamurthy[◇] Scott Shenker[†⋆]
[†]UC Berkeley [‡]Tsinghua University [◇]University of Washington [⋆]ICSI

Abstract

We consider the following question: *what consistency model is appropriate for coordinating the actions of a replicated set of SDN controllers?* We first argue that the conventional requirement of strong consistency, typically achieved through the use of Paxos or other consensus algorithms, is conceptually unnecessary to handle unplanned network updates. We present an alternate approach, based on the weaker notion of eventual correctness, and describe the design of a simple coordination layer (SCL) that can seamlessly turn a set of single-image SDN controllers (that obey certain properties) into a distributed SDN system that achieves this goal (whereas traditional consensus mechanisms do not). We then show through analysis and simulation that our approach provides faster responses to network events. While our primary focus is on handling unplanned network updates, our coordination layer also handles policy updates and other situations where consistency is warranted. Thus, contrary to the prevailing wisdom, we argue that distributed SDN control planes need only be slightly more complicated than single-image controllers.

1 Introduction

Software-Defined Networking (SDN) uses a "logically centralized" controller to compute and instantiate forwarding state at all switches and routers in a network. However, behind the simple but ambiguous phrase "logically centralized" lies a great deal of practical complexity. Typical SDN systems use multiple controllers to provide availability in the case of controller failures.[1] However, ensuring that the behavior of replicated controllers matches what is produced by a single controller requires coordination to ensure consistency. Most distributed SDN controller designs rely on consensus mechanisms such as Paxos (used by ONIX [14]) and Raft (used by ONOS [2]), and recent work (*e.g.,* Ravana [10]) require even stronger consistency guarantees.

But consistency is only a means to an end, not an end in itself. Operators and users care that certain properties or invariants are obeyed by the forwarding state installed in switches, and are not directly concerned about consistency among controllers. For example, they care whether the forwarding state enforces the desired isolation between hosts (by installing appropriate ACLs), or enables the desired connectivity between hosts (by installing functioning paths), or establishes paths that traverse a specified set of middleboxes; operators and users do not care whether the controllers are in a consistent state when installing these forwarding entries.

With this invariant-oriented criterion in mind, we revisit the role of consistency in SDN controllers. We analyze the consistency requirements for the two kinds of state that reside in controllers — *policy state* and *network state*[2] — and argue that for network state consensus-based mechanisms are both *conceptually inappropriate* and *practically ineffective*. This raises two questions: why should we care, and what in this argument is new?

Why should we care? Why not use the current consistency mechanisms even if they are not perfectly suited to the task at hand? The answer is three-fold.

First, consistency mechanisms are both algorithmically complex and hard to implement correctly. To note a few examples, an early implementation of ONIX was plagued by bugs in its consistency mechanisms; and people continue to find both safety and liveness bugs in Raft [6, 22]. Consistency mechanisms are among the most complicated aspects of distributed controllers, and should be avoided if not necessary.

Second, consistency mechanisms restrict the availability of systems. Typically consensus algorithms are available only when a majority of participants are active and connected. As a result consistency mechanisms prevent distributed controllers from making progress under severe failure scenarios, *e.g.,* in cases where a partition is only accessible by a minority of controllers. Consistency and availability are fundamentally at odds during such failures, and while the lack of availability may be necessary for some policies, it seems unwise to pay this penalty in cases where such consistency is not needed.

Third, consistency mechanisms impose extra latency in responding to events. Typically, when a link fails, a

[1] In some scenarios multiple controllers are also needed to scale controller capacity (*e.g.,* by sharding the state between controllers), but in this paper we focus on replication for reliability.

[2] As we clarify later, network state describes the current network topology while policy state describes the properties or invariants desired by the operator (such as shortest path routing, or access control requirements, or middlebox insertions along paths).

switch sends a notification to the nearest controller, which then uses a consensus protocol to ensure that a majority of controllers are aware of the event and agree about when it occurred, after which one or more controllers change the network configuration. While the first step (switch contacting a nearby controller) and last step (controllers updating switch configuration) are inherent in the SDN control paradigm, the intervening coordination step introduces extra delay. While in some cases – *e.g.,* when controllers reside on a single rack – coordination delays may be negligible, in other cases – *e.g.,* when controllers are spread across a WAN – coordination can significantly delay response to failures and other events.

What is new here? Much work has gone into building distributed SDN systems (notably ONIX, ONOS, ODL, and Ravana),[3] and, because they incorporate sophisticated consistency mechanisms, such systems are significantly more complex than the *single-image controllers* (such as NOX, POX, Beacon, Ryu, etc.) that ushered in the SDN era.[4] In contrast, our reconsideration of the consistency requirements (or lack thereof) for SDN led us to design a simple coordination layer (SCL) that can transform any single-image SDN controller design into a distributed SDN system, as long as the controller obeys a small number of constraints.[5] While our novelty lies mostly in how we handle unplanned updates to network state, SCL is a more general design that deals with a broader set of issues: different kinds of network changes (planned and unplanned), changes to policy state, and the consistency of the data plane (the so-called consistent updates problem). All of these are handled by the coordination layer, leaving the controller completely unchanged. *Thus, contrary to the prevailing wisdom, we argue that distributed SDN systems need only be slightly more complicated than single-image controllers.* In fact, they can not only be simpler than current distributed SDN designs, but have better performance (responding to network events more quickly) and higher availability (not requiring a majority of controllers to be up at all times).

2 Background

In building SDN systems, one must consider consistency of both the data and control planes. Consider the case where a single controller computes new flow entries for all switches in the network in response to a policy or network update. The controller sends messages to each switch updating their forwarding entries, but these updates are applied asynchronously. Thus, until the last switch has received the update, packets might be handled by a mixture of updated and non-updated switches, which could lead to various forms of invariant violations (*e.g.,* looping paths, missed middleboxes, or lack of isolation). The challenge, then, is to implement these updates in a way that no invariants are violated; this is often referred to as the *consistent updates* problem and has been addressed in several recent papers [11, 17, 20, 21, 25, 26].

There are three basic approaches to this problem. The first approach carefully orders the switch updates to ensure no invariant violations [17, 20]. The second approach tags packets at ingress, and packets are processed based on the new or old flow entries based on this tag [11, 25, 26]. The third approach relies on closely synchronized switch clocks, and has switches change over to the new flow entries nearly-simultaneously [21].

Note that the consistent updates problem exists even for a single controller and is not caused by the use of replicated controllers (which is our focus); hence, we do not introduce any new mechanisms for this problem, but can leverage any of the existing approaches in our design. In fact, we embed the tagging approach in our coordination layer, so that controllers need not be aware of the consistent updates problem and can merely compute the desired flow entries.

The problem of control plane consistency arises when using replicated controllers. It seems natural to require that the state in each controller – *i.e.,* their view of the network and policy – be consistent, since they must collaboratively compute and install forwarding state in switches, and inconsistency at the controller could result in errors in forwarding entries. As a result existing distributed controllers use consensus algorithms to ensure serializable updates to controller state, even in the presence of failures. Typically these controllers are assumed to be deterministic – *i.e.,* their behavior depends only on the state at a controller – and as a result consistency mechanisms are not required for the output. Serializability requires coordination, and is typically implemented through the use of consensus algorithms such as Paxos and Raft. Commonly these algorithms elect a leader from the set of available controllers, and the leader is responsible for deciding the order in which events occur. Events are also replicated to a quorum (typically a majority) of controllers before any controller responds to an event. Replication to a quorum ensures serializability even in cases where the leader fails, this is because electing a new leader requires use of a quorum that intersect with all previous quorums [7].

More recent work, *e.g.,* Ravana [10], has looked at requiring even stronger consistency guarantees. Ravana

[3] There are many other distributed SDN platforms, but most (such as [19, 28]) are focused on sharding state in order to scale (rather than replicating for availability), which is not our focus. Note that our techniques can be applied to these sharded designs (*i.e.,* by replicating each shard for reliability).

[4] By the term "single-image" we mean a program that is written with the assumption that it has unilateral control over the network, rather than one explicitly written to run in replicated fashion where it must coordinate with others.

[5] Note that our constraints make it more complex to deal with policies such as reactive traffic engineering that require consistent computation on continuously changing network state.

tries to ensure exactly-once semantics when processing network events. This stronger consistency requirement comes at the cost of worse availability, as exactly-once semantics require that the system be unavailable (*i.e.,* unresponsive) in the presence of failures [15].

While the existing distributed controller literature varies in mechanisms (*e.g.,* Paxos, Raft, ZAB, etc.) and goals (from serializability to exactly-once semantics) there seems to be universal agreement on what we call the *consensus assumption*; that is the belief that consensus is the weakest form of coordination necessary to achieve correctness when using replicated controllers, *i.e.,* controllers must ensure serializability or stronger consistency for correctness. The consensus assumption follows naturally from the concept of a "logically centralized controller" as serializability is required to ensure that the behavior of a collection of replicated controllers is identical to that of a single controller.

However, we do not accept the consensus assumption, and now argue that *eventual correctness* – which applies after controllers have taken action – not *consensus* is the most salient requirement for distributed controllers. Eventual correctness is merely the property that in any connected component of the network which contains one or more controller, all controllers eventually converge to the correct view of the network, *i.e.,* in the absence of network updates all controllers will eventually have the correct view of the network (*i.e.,* its topology and configuration) and policy, and that the forwarding rules installed within this connected component will all be computed relative to this converged network view and policy. This seems like a weaker property than serializability, but cannot be achieved by consensus based controllers which require that a quorum of controllers be reachable.

So why are consensus-based algorithms used so widely? Traditional systems (such as data stores) that use consensus algorithms are "closed world", in that the truth resides within the system and no update can be considered complete until a quorum of the nodes have received the update; otherwise, if some nodes fail the system might lose all memory of that update. Thus, no actions should be taken on an update until consensus is reached and the update firmly committed. While policy state is closed-world, in that the truth resides in the system, network state is "open-world" in that the ground truth resides in the network itself, not in the controllers, *i.e.,* if the controllers think a link is up, but the link is actually down, then the truth lies in the functioning of the link, not the state in the controller. One can always reconstruct network state by querying the network. Thus, one need not worry about the controllers "forgetting" about network updates, as the network can always remind them. This removes the need for consensus *before* action, and the need for timeliness would suggest acting without this additional delay.

To see this, it is useful to distinguish between *agreement* (do the controllers *agree* with each other about the network state?) and *awareness* (is at least one controller *aware* of the current network state?). If networks were a closed-world system, then one should not update the dataplane until the controllers are in agreement, leading to the consensus assumption. However, since networks are an open-world system, updates can and should start as soon as any controller is aware, without waiting for agreement. Waiting for agreement is unnecessary (since network state can always be recovered) and undesirable (since it increases response time, reduces availability, and adds complexity).

Therefore, SDN controllers should not unnecessarily delay updates while waiting for consensus. However, we should ensure that the network is eventually correct, *i.e.,* controllers should eventually agree on the current network and policy state, and the installed forwarding rules should correctly enforce policies relative to this state. The rest of this paper is devoted to describing a design that uses a simple coordination layer lying underneath any single-image controller (that obeys certain constraints) to achieve rapid and robust responses to network events, while guaranteeing eventual correctness. Our design also includes mechanisms for dataplane consistency and policy consistency.

3 Definitions and Categories

3.1 Network Model

We consider networks consisting of switches and hosts connected by *full-duplex links*, and controlled by a set of replicated controllers which are responsible for configuring and updating the forwarding behavior of all switches. As is standard for SDNs, we assume that switches notify controllers about network events, *e.g.,* link failures, when they occur. Current switches can send these notifications by using either a separate control network (out-of-band control) or using the same networks as the one being controlled (in-band control). In the rest of this paper we assume the use of an in-band control network, and use this to build *robust channels* that guarantee that the controller can communicate with all switches within its partition, *i.e.,* if a controller can communicate with some switch A, then it can also communicate with any switch B that can forward packets to A. We describe our implementation of robust channels in §6.1. We also assume that the control channel is fair – *i.e.,* a control message sent infinitely often is delivered to its destination infinitely often – this is a standard assumption for distributed systems.

We consider a *failure model* where any controller, switch, link or host can fail, and possibly recover after an arbitrary delay. Failed network component stop functioning, and no component exhibits Byzantine behavior. We further assume that when alive controllers and switches are responsive – *i.e.,* they act on all received

messages in bounded time – this is true for existing switches and controllers, which have finite queues. We also assume that all active links are full-duplex, *i.e.,* a link either allows bidirectional communication or no communication. Certain switch failures can result in asymmetric link failures – *i.e.,* result in cases where communication is only possible in one direction – this can be detected through the use of Bidirection Forwarding Detection (BFD) [12, 13] at which point the entire link can be marked as having failed. BFD is implemented by most switches, and this functionality is readily available in networks today. Finally we assume that the failure of a switch or controller triggers a network update – either by a neighboring switch or by an operator, the mechanism used by operators is described in Appendix B.

We define *dataplane configuration* to be the set of forwarding entries installed at all functioning switches in the network; and *network state* as the undirected graph formed by the functioning switches, hosts, controllers and links in a network. Each edge in the network state is annotated with relevant metadata about link properties – *e.g.,* link bandwidth, latency, etc. The *network configuration* represents the current network state and dataplane configuration. A *network policy* is a predicate over the network configuration: a policy holds if and only if the predicate is true given the current network configuration. Network operators configure the network by specifying a set of network policies that should hold, and providing mechanisms to restore policies when they are violated.

Given these definitions, we define a network as being *correct* if and only if it implements all network policies specified by the operator. A network is rendered *incorrect* if a policy predicate is no longer satisfied as a result of one or more *network events*, which are changes either to network state or network policy. Controllers respond to such events by modifying the dataplane configuration in order to restore correctness.

Controllers can use a *dataplane consistency mechanism* to ensure dataplane consistency during updates. The controllers also use a *control plane consistency mechanism* to ensure that after a network event controllers eventually agree on the network configuration and that the data plane configuration converges to a correct configuration, *i.e.,* one that is appropriate to the current network state and policy.

Consistency mechanisms can be evaluated based on how they ensure correctness following a network event. Their design is thus intimately tied to the nature of *policies* implemented by a network and the types of *events* encountered, which we now discuss.

3.2 Policy Classes

Operators can specify *liveness policies* and *safety policies* for the network. Liveness policies are those that must hold in steady state, but can be violated for short periods of time while the network responds to network events. This is consistent with the definition of *liveness* commonly used in distributed systems [16]. Sample liveness properties include:

1. *Connectivity*: requiring that the data plane configuration is such that packets from a source are delivered to the appropriate destination.
2. *Shortest Path Routing*: requiring that all delivered packets follow the shortest path in the network.
3. *Traffic Engineering*: requiring that for a given traffic matrix, routes in the network are such that the maximum link utilization is below some target.

Note that all of these examples require some notion of global network state, *e.g.,* one cannot evaluate whether a path is shortest unless one knows the entire network topology. Note that such policies are inherently liveness properties – as discussed below, one cannot ensure that they always hold given the reliance on global state.

A *safety policy* must always hold, regardless of any sequence of network events. Following standard impossibility results in distributed systems [23], a policy is enforceable as a safety policy in SDN if and only if it can be expressed as a condition on a path – *i.e.,* a path either obeys or violates the safety condition, regardless of what other paths are available. Safety properties include:

1. *Waypointing*: requiring that all packets sent between a given source and destination traverse some middlebox in a network.
2. *Isolation*: requiring that some class of packets is never received by an end host (such as those sent from another host, or with a given port number).

In contrast to liveness properties that require visibility over the entire network, these properties can be enforced using information carried by each packet using mechanisms borrowed from the consistent updates literature. Therefore, we extend the tagging approach from the consistent updates literature to implement safety policies in SCL. Our extension, and proofs showing that this is both necessary and sufficient, are presented in Appendix A.

3.3 Network Events

Next we consider the nature of network events, focusing on two broad categories:

1. *Unplanned network events*: These include events such as link failures and switch failures. Since these events are unplanned, and their effects are observable by users of the network, it is imperative that the network respond quickly to restore whatever liveness properties were violated by these topology events. Dealing with these events is the main focus of our work, and we address this in §5.
2. *Policy Changes*: Policy state changes slowly (*i.e.,* at human time scales, not packet time scales), and perfect *policy availability* is not required, as one can temporarily

block policy updates without loss of *network availability* since the network would continue operating using the previous policy. Because time is not of the essence, we use two-phase commit when dealing with policy changes, which allows us to ensure that the policy state at all controllers is always consistent. We present mechanisms and analysis for such events in Appendix B.

Obviously not all events fall into these two categories, but they are broader than they first appear. Planned network events (*e.g.,* taking a link or switch down for maintenance, restoring a link or switch to service, changing link metadata) should be considered policy changes, in that these events are infrequent and do not have severe time constraints, so taking the time to achieve consistency before implementing them is appropriate. Load-dependent policy changes (such as load balancing or traffic engineering) can be dealt with in several ways: (i) as a policy change, done periodically at a time scale that is long compared to convergence time for all distributed consistency mechanisms (*e.g.,* every five minutes); or (ii) using dataplane mechanism (as in MATE [3]) where the control plane merely computes several paths and the dataplane mechanism responds to load changes in real time without consulting the control plane. We provide a more detailed discussion of traffic engineering in Appendix B.2.

3.4 The Focus of Our Work

In this paper, we focus on how one can design an SDN control plane to handle unplanned network events so that liveness properties can be more quickly restored. However, for completeness, we also present (but claim no novelty for) mechanisms that deal with policy changes and safety properties. In the next section, we present our design of SCL, followed by an analysis (in Section 5) of how it deals with liveness properties. In Section 6 we describe SCL's implementation, and in Section 7 present results from both our implementation and simulations on SCL's performance. We delay until the appendices discussion of how SCL deals with safety properties.

4 SCL Design

SCL acts as a coordination layer for single-image controllers (*e.g.,* Pox, Ryu, etc.). Single image controllers are designed to act as the sole controller in a network, which greatly simplifies their design, but also means that they cannot survive any failures, which is unacceptable in production settings. SCL allows a single image controller to be replicated on multiple physical controllers thereby forming a distributed SDN control plane. We require that controllers and applications used with SCL meet some requirements (§4.1) but require no other modifications.

4.1 Requirements

We impose four requirements on controllers and applications that can be used with SCL:

(a) **Deterministic**: We require that controller applications be deterministic with respect to network state. A similar requirement also applies to RSM-based distributed controllers (*e.g.,* ONOS, Onix).

(b) **Idempotent Behavior**: We require that the commands controllers send switches in response to any events are idempotent, which is straightforward to achieve when managing forwarding state. This requirement matches the model for most OpenFlow switches.

(c) **Triggered Recomputation**: We require that on receiving an event, the controller recomputes network state based on a log containing the sequence of network events observed thus far. This requires that the controller itself not retain state and incrementally update it, but instead use the log to regenerate the state of the network. This allows for later arriving events to be inserted earlier in the log without needing to rewrite internal controller state.

(d) **Proactive Applications**: We require that controller applications compute forwarding entries based on their picture of the network state. We do not allow reactive controller applications which respond to individual packet-ins (such as might be used if one were implementing a NAT on a controller).

In addition to the requirements imposed on controller applications, SCL also requires that all control messages – including messages used by switches to notify controllers, messages used by controllers to update switches, and messages used for consistency between controllers – be sent over *robust* communication channels. We define a *robust* communication channel as one which ensures connectivity as long as the network is not partitioned – *i.e.,* a valid forwarding path is always available between any two nodes not separated by a a network partition.

4.2 General Approach

We build on existing single-image controllers that recompute dataplane configuration from a log of events each time recomputation is triggered (as necessitated by the triggered recomputation requirement above). To achieve eventual correctness, SCL must ensure – assuming no further updates or events – that in any network partition containing a controller eventually (i) every controller has the same log of events, and (ii) this log accurately represents the current network state (within the partition). In traditional distributed controllers, these requirements are achieved by assuming that a quorum of controllers (generally defined to be more than one-half of all controllers) can communicate with each other (*i.e.,* they are both functioning correctly and within the same partition), and using a consensus algorithm to commit events to a quorum before they are processed. SCL ensures these requirements – without the restriction on having a

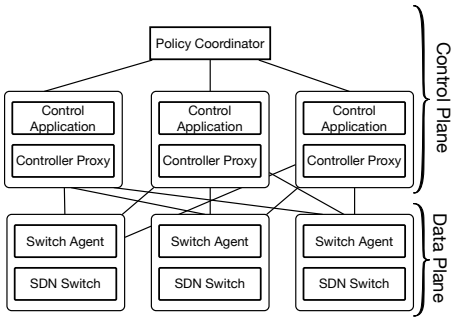


Figure 1: Components in SCL.

controller quorum – through the use of two mechanisms:

Gossip: All live controllers in SCL periodically exchange their current event logs. This ensures that eventually the controllers agree on the log of events. However, because of our failure model, this gossip cannot by itself guarantee that logs reflect the current network state.

Periodic Probing: Even though switches send controllers notifications when network events (*e.g.,* link failures) occur, a series of controller failures and recovery (or even a set of dropped packets on links) can result in situations where no live controller is aware of a network event. Therefore, in SCL all live controllers periodically send probe messages to switches; switches respond to these probe messages with their set of working links and their flow tables. While this response is not enough to recover all lost network events (*e.g.,* if a link fails and then recovers before a probe is sent then we will never learn of the link failure), this probing mechanism ensures eventual awareness of the current network state and dataplane configuration.

Since we assume *robust* channels, controllers in the same partition will eventually receive gossip messages from each other, and will eventually learn of all network events within the same partition. We *do not assume control channels are lossless*, and our mechanisms are designed to work even when messages are lost due to failures, congestion or other causes. Existing SDN controllers rely on TCP to deal with losses.

Note that we explicitly design our protocols to ensure that controllers never disagree on policy, as we specify in greater detail in Appendix B. Thus, we do not need to use gossip and probe mechanisms for policy state.

4.3 Components

Figure 1 shows the basic architecture of SCL. We now describe each of the components in SCL, starting from the data plane.

SDN Switches: We assume the use of *standard SDN switches*, which can receive messages from controllers and update their forwarding state accordingly. We make no specific assumptions about the nature of these messages, other than assuming they are idempotent. Furthermore, we require no additional hardware or software support beyond the ability to run a small proxy on each switch, and support for BFD for failure detection. As discussed previously, BFD and other failure detection mechanisms are widely implemented by existing switches. Many existing SDN switches (*e.g.,* Pica-8) also support replacing the OpenFlow agent (which translates control messages to ASIC configuration) with other applications such as the switch-agent. In case where this is not available, the proxy can be implemented on a general purpose server attached to each switch's control port.

SCL switch-agent: The switch-agent acts as a proxy between the control plane and switch's control interface. The switch-agent is responsible for implementing SCL's robust control plane channels, for responding to periodic probe messages, and for forwarding any switch notifications (*e.g.,* link failures or recovery) to all live controllers in SCL. The switch-agent, with few exceptions, immediately delivers any flow table update messages to the switch, and is therefore not responsible for providing any ordering or consistency semantics.

SCL controller-proxy: The controller-proxy implements SCL's consistency mechanisms for both the control and data plane. To implement control plane consistency, the controller-proxy (a) receives all network events (sent by the switch-agent), (b) periodically exchanges gossip messages with other controllers; and (c) sends periodic probes to gain awareness of the dataplane state. It uses these messages to construct a consistently ordered log of network events. Such a consistently ordered log can be computed using several mechanisms, *e.g.,* using accurate timestamps attached by switch-agents. We describe our mechanism—which relies on switch IDs and local event counters—in §6.2. The controller-proxy also keeps track of the network's dataplane configuration, and uses periodic-probe messages to discover changes. The controller-proxy triggers recomputation at its controller whenever it observes a change in the log or dataplane configuration. This results in the controller producing a new dataplane configuration, and the controller-proxy is responsible for installing this configuration in the dataplane. SCL implements data plane consistency by allowing the controller-proxy to transform this computed dataplane configuration before generating switch update messages as explained in Appendix A.

Controller: As we have already discussed, SCL uses standard single image controllers which must meet the four requirements in §4.1.

These are the basic mechanisms (gossip, probes) and components (switches, agents, proxies, and controllers) in SCL. Next we discuss how they are used to restore *liveness policies* in response to *unplanned topology updates*. We later generalize these mechanisms to allow

handling of other kinds of events later in Appendix B.

5 Liveness Policies in SCL

We discuss how SCL restores *liveness policies* in the event of *unplanned topology updates*. We begin by presenting SCL's mechanisms for handling such events, and then analyze their correctness.

5.1 Mechanism

Switches in SCL notify their switch-agent of any network events, and switch-agents forward this notification to all controller-proxies in the network (using the control plane channel discussed in §6). Updates proceed as follows once these notifications are sent:

- On receiving a notification, each controller-proxy updates its event log. If the event log is modified[6], the proxy triggers a recomputation for its controller.
- The controller updates its network state, and computes a new dataplane configuration based on this updated state. It then sends a set of messages to install the new dataplane configuration which are received by the controller-proxy.
- The controller-proxy modifies these messages appropriately so safety policies are upheld (Appendix A) and then sends these messages to the appropriate switch switch-agents.
- Upon receiving an update message, each switch's switch-agent immediately (with few exceptions discussed in Appendix B) updates the switch flow table, thus installing the new dataplane configuration.

Therefore, each controller in SCL responds to unplanned topology changes without coordinating with other controllers; all the coordination happens in the coordination layer, which merely ensures that each controller sees the same log of events. We next show how SCL achieves correctness with these mechanisms.

5.2 Analysis

Our mechanism needs to achieve eventual correctness, *i.e.,* after a network event and in the absence of any further events, there must be a point in time after which all policies hold forever. This condition is commonly referred to as *convergence* in the networking literature, which requires a quiescencent period (*i.e.,* one with no network events) for convergence. Note that during quiescent periods, no new controllers are added because adding a controller requires either a network event (when a partition is repaired) or a policy change (when a new controller is inserted into the network), both of which violate the assumption of quiescence. We observe that *eventual correctness* is satisfied once the following properties hold during the quiescent period:

[6]Since controller-proxies update logs in response to both notifications from switch-agents and gossip from other controllers, a controller-proxy may be aware of an event before the notification is delivered.

i. The control plane has reached *awareness – i.e.,* at least one controller is aware of the current network configuration.
ii. The controllers have reached *agreement – i.e.,* they agree on network and policy state.
iii. The dataplane configuration is correct – *i.e.,* the flow entries are computed with the correct network and policy state.

To see this consider these conditions in order. Because controllers periodically probe all switches, no functioning controller can remain ignorant of the current network state forever. Recall that we assume that switches are responsive (while functioning, if they are probed infinitely often they respond infinitely often) and the control channel is fair (if a message is sent infinitely often, it is delivered infinitely often), so an infinite series of probes should produce an infinite series of responses. Once this controller is aware of the current network state, it cannot ever become unaware (that is, it never forgets the state it knows, and there are no further events that might change this state during a quiscent period). Thus, once condition 1 holds, it will continue to hold during the quiescent period.

Because controllers gossip with each other, and also continue to probe network state, they will eventually reach agreement (once they have gossiped, and no further events arrive, they agree with each other). Once the controllers are in agreement with each other, and with the true network state, they will never again disagree in the quiescent period. Thus, once conditions 1 and 2 hold, they will continue to hold during the quiescent period.

Similarly, once the controllers agree with each other and reality, the forwarding entries they compute will eventually be inserted into the network – when a controller probes a switch, any inconsistency between the switch's forwarding entry and the controllers entry will result in an attempt to install corrected entries. And once forwarding entries agree with those computed based on the true network state, they will stay in agreement throughout the quiescent period. Thus, once conditions 1, 2, and 3 hold, they will continue to hold during the quiescent period. This leads to the conclusion that once the network becomes quiescent it will eventually become correct, with all controllers aware of the current network state, and all switches having forwarding entries that reflect that state.

We now illustrate the difference between SCL's mechanisms and consensus based mechanisms through the use of an example. For ease of exposition we consider a small network with 2 controllers (quorum size is 2) and 2 switches, however our arguments are easily extended to larger networks. For both cases we consider the control plane's handling of a single link failure. We assume that the network had converged prior to the link failure.

First, we consider SCL's handling of such an event, and show a plausible timeline for how this event is handled in

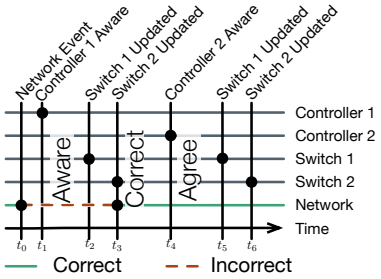


Figure 2: Response to an unplanned topology event in SCL.

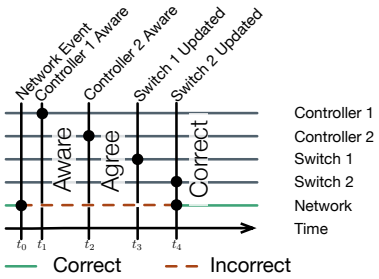


Figure 3: Response to an unplanned topology event when using consensus based controllers.

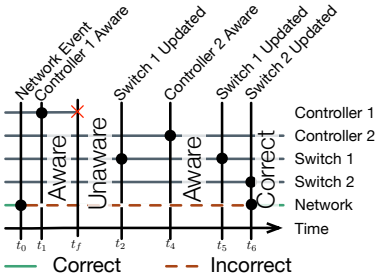


Figure 4: Response to one unplanned topology event in the presence of controller failure.

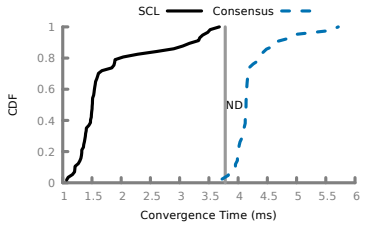


Figure 5: Fat Tree: CDF of time taken for routes to converge after single link failure. There are cases that exceed ND because of queuing in the network.

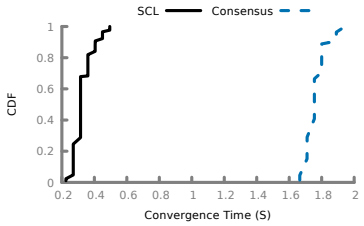


Figure 6: AS1221: CDF of time taken for routes to converge after single link failure.

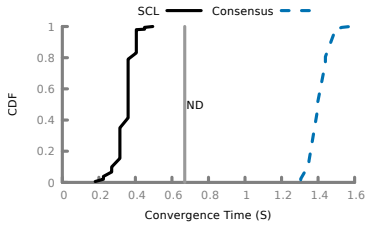


Figure 7: AS1239: Time taken for routes to converge after single link failure.

Figure 2. At time t_0 a network event occurs which renders the network incorrect (some liveness properties are violated) and the control plane unaware. At time t_1 Controller 1 receives a notification about the event, this makes the control plane *aware* of the event, but the control plane is no longer in *agreement*, since Controller 2 is unaware. Controller 1 immediately computes updates, which are sent to the switch. Next at time t_3 both switches have received the updated configuration, rendering the network *correct*. Finally, at time t_4, Controller 2 receives notification of this event, rendering the control plane in *agreement*. The network is thus converged at t_4. Note, since SCL does not use consensus, this is not the only timeline possible in this case. For example, another (valid) timeline would have the controllers reaching *agreement* before the dataplane has reached *correctness*. Finally, we also note that even in cases where a notification is lost, gossip would ensure that the network will reach agreement in this case.

Next, we show how such an event is handled in a consensus based controller framework in Figure 3. Without loss of generality, we assume that Controller 1 is the leader in this case. Similar to above, the control plane becomes aware at time t_1. Next, at time t_2, Controller 2 becomes aware of the event through the consensus mechanism, the controllers therefore reach *agreement*. Once agreement has been reached Controller 1 computes updates, and the network is rendered correct at time t_4. Consensus mechanisms require that any configuration changes occur after agreement, hence this is the only plausible timeline in this case. Finally, observe that when consensus mechanisms are used the dataplane configuration is updated once (at t_3 and t_4), while SCL issues two sets of updates, one from each controller. This is a necessary consequence of our design.

Finally, we consider a case where a controller, *e.g.,* Controller 1, fails while the event is being handled. Because we assumed that our network had 2 controllers, consensus algorithms cannot make progress in this case. We therefore focus on SCL's response, and show a timeline in Figure 4. In this case the control plane is temporarily aware of the network event at time t_1, but becomes unaware when Controller 1 fails at time t_f. However, since switches notify all live controllers, Controller 2 eventually receives the notification at time t_4, rendering the control plane *aware* and in *agreement*. At this time Controller 2 can generate updates for both Switch 1 and 2 rendering the network correct. Note, that even in cases where the Controller 2's notification is lost (*e.g.,* due to message losses), it would eventually be aware due to periodic probing.

6 Implementation

Our implementation of SCL uses Pox [18], a single-image controller. To simplify development of control applications, we changed the *discovery* and *topology* modules in POX so they would use the log provided by the controller-proxy in SCL. We also implemented a switch-agent that works in conjunction with OpenVSwitch [24], a popular software switch. In this section we present a few implementation details about how we implement robust channels (§6.1); how we ensure log consistency across controller-proxies (§6.2); and a few optimizations that we found important for our implementation (§6.3).

6.1 Robust Channel

Most existing SDN implementations require the use of a separate control network (referred to as out-of-band

control). In contrast, our implementation of SCL reuses the same network for both control and data packets. The use of such an *in-band control* network is essential for implementing *robust* channels required by SCL (where we assume the control channel functions between any two connected nodes). In our design, the switch-agent is responsible for implementing control channels. It does so by first checking if it has previously seen a received message, and flooding (*i.e.,* broadcasting) the control message if it has not. Flooding the message means that each message goes through all available paths in the network, ensuring that in the absence of congestion all control messages are guaranteed to be delivered to all nodes in the same network partition. This meets our requirements for *robust channels* from §4.2. We show in the evaluation that this broadcast-based robust channel consumes a small percentage ($<$ 1Mbps) of the network bandwidth. Furthermore, since SCL's correctness does not depend on reliable delivery of control plane messages, we also limit the amount of bandwidth allocated for control messages (by prioritizing them) and rely on switches to drop any traffic in excess of this limit, so that even under extreme circumstances the control traffic is limited to a small fraction of the total bandwidth.

6.2 Ordering Log Messages

Ensuring that controllers reach agreement requires the use of a mechanism to ensure that event ordering at each controller is eventually the same (§5). In our analysis we assumed the use of some ordering function that used metadata added by switch-agents to network events to produce a total order of events that all controllers would agree on. Here we describe that mechanism in greater detail. Switch agents in SCL augment network events with a *local* (*i.e.,* per switch-agent) sequence number. This sequence number allows us to ensure that event ordering in controller logs always corresponds to the causal order at each switch; we call this property *local causality*. Switch agent failures might result in this sequence number being reset, we therefore append an epoch count (which is updated when a switch boots) to the sequence number, and ensure that sequence numbers from the same switch-agent are monotically increasing despite failures. We assume the epoch is stored in stable storage and updated each time a switch boots. We also assume that each switch is assigned an ID, and that switch IDs impose a *total order* on all switches. IDs can be assigned based on MAC addresses, or other mechanisms.

Our algorithm for ensuring consistent ordering depends on the fact that *gossip messages* include the position of each event in the sender's log. Our algorithm then is simple: when notified of an event by the data plane (this might be due to a direct event notification or because of periodic probing), each controller-proxy inserts the event at the end of the log. If such an insertion violates local causality, SCL swaps events until local causality is restored, *e.g.,* if a controller is notified of an event e from a switch s with sequence number 5, but the log already contains another event e' from the same switch with sequence number 6, then these events are swapped so that e occupies the log position previously occupied by e' and e' appears at the end of the log. It is simple to see that the number of swaps needed is bounded. When a controller-proxy receives a gossip message, it checks to see if its position for event disagrees with the position for the sender of the gossip message. If there is a disagreement, we swap messages to ensure that the message sent by a switch with a lower ID appears first. Again it is easy to see that the number of swaps required is bounded by the number of positions on which the logs disagree.

Assuming that the network being controlled goes through periods of change (where a finite number of new network events occur) followed by periods of quiescence (where no new network events occur), and that quiescent periods are long enough to reach agreement, then the only events reordered are those which occur within a single period of change. Since we assumed that only a finite number of new events occur within a period of change, there's only a finite number of event swaps that need to be carried out to ensure that the log ordering is the same across all controllers. Furthermore, any event that is not in its final position must be swapped during a round of gossip. This is because otherwise all controllers must agree on the message's position, and hence it will never subsequently be changed. This implies that all controllers must agree upon log ordering within a finite number of gossip rounds. Note our assumption about quiescent periods is required by all routing algorithms, and is not unique to SCL.

6.3 Optimizations

Our implementation also incorporates three optimizations to reduce bandwidth usage. Our optimizations generally aim to reduce the size of messages, but occasionally lead to an increase in the number of messages.

First, we observe that each switch's response to periodic probes includes a copy of its flow tables along with link state information. Switch flow table sizes can range from several 1000-100K flow entries, and in a naive implementation these messages would rapidly consume most of the bandwidth available for control channels. However, in the absence of controller failures (*i.e.,* in the common case) it is unlikely that a controller's version of the switch's flow table differs from reality. Therefore, when sending a periodic probe request to a switch s, the controller's controller-proxy includes the hash of their view of s's flow tables in the request. The switch-agents sends a flow table as a part of their response if and only if this hash differs from the hash for the actual flow table.

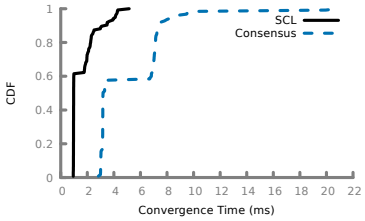


Figure 8: Fat Tree: CDF of time taken for the network (switches and controllers) to fully agree after a single link failure.

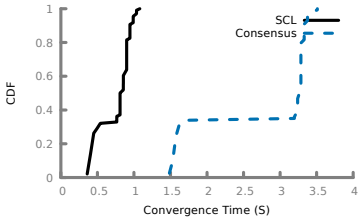


Figure 9: AS1221: CDF of time taken for the network (switches and controllers) to fully agree after a single link failure.

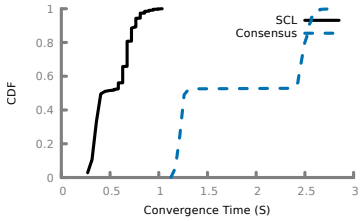


Figure 10: AS1239: CDF of time taken for the network (switches and controllers) to fully agree after a single link failure.

This dramatically reduces the size of probe responses in the common case.

Second, we require that controller-proxies converge on a single network event log. However, here again sending the entire log in each Gossip message is impractical. To address this we use a log truncation algorithm, whereby switch-agents no longer include log entries about which all active controllers agree. This works as follows: (a) controller-proxies always include new or modified log entries (since a previous Gossip round, *i.e.,* the last time all controllers exchanged gossip messages and agreed on log ordering) in gossip message; (b) any time a controller-proxy finds that all active controllers agree on a log entry it marks that entry as committed and stops including it in the log; and (c) controller-proxies can periodically send messages requesting uncommitted log entries from other controllers. The final mechanism (requesting logs) is required when recovering from partitions.

Log truncation is also necessary to ensure that the controller-proxy does not use more than a small amount of data. The mechanism described above suffices to determine when all active controllers are aware of a log message. controller-proxies periodically save committed log messages to stable storage allowing them to free up memory. New controllers might require these committed messages, in which case they can be fetched from stable storage. Committed messages can also be permanently removed when controller applications no longer require them, however this requires awarness of application semantics and is left to future work. Finally, since log compaction is not essential for correctness we restrict compaction to periods when the all controllers in a network are in the same partition, *i.e.,* all controllers can communicate with each other – this greatly simplifies our design.

Finally, similar to other SDN controllers, SCL does not rewrite switch flow tables and instead uses incremental flow updates to modify the tables. Each controller-proxy maintains a view of the current network configuration, and uses this information to only send updates when rules need to change. Note, in certain failure scenarios, *e.g.,* when a controller fails and another simultaneously recovers, this might incur increased time for achieving correctness, however our correctness guarantees still hold.

As we show later in §7 these optimizations are enough to ensure that we use no more than 1Mbps of bandwidth for control messages in a variety of networks.

7 Evaluation

As mentioned earlier, the standard approach to replicating SDN controllers is to use consensus algorithms like Paxos and Raft to achieve consistency among replicas. The design we have put forward here eschews such consensus mechanisms and instead uses mostly-unmodified single-image controllers and a simple coordination layer. Our approach has two clear advantages:

Simplicity: Consistency mechanisms are complicated, and are often the source of bugs in controller designs. Our approach is quite simple, both conceptually and algorithmically, leading to a more robust overall design whose properties are easy to understand.

Eventual Correctness: This seems like the most basic requirement one could have for distributed controllers, and is easily achieved in our design, yet consensus mechanisms violate it in the presence of partitions.

Given these advantages, one might ask why might one choose current consensus-based approaches over the one we have presented here? We investigate three possible reasons: response times, impact of controller disagreement, and bandwidth overhead. Our analysis uses the network model presented in §7.1, which allows us to quantitatively reason about response times. We then use simulations to gain a deeper understanding of how the design operates. We use simulation rather than our implementation because we we can more closely control the environment and monitor the results.

Finally, in §7.4 we use CloudLab [1] to compare convergence times for SCL and ONOS [2] when responding to link failures in a datacenter topology. Our results demonstrate that the effects observed in simulation also hold for our implementation.

7.1 Response to Network Events

The primary metric by which SDN controller designs should be evaluated is the extent to which they achieve

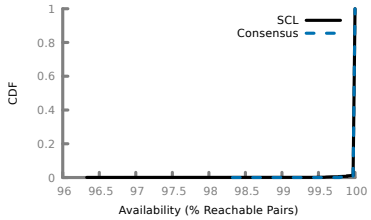


Figure 11: Fat Tree: CDF of availability showing percentage of physically connected host pairs that are reachable based on routing rules.

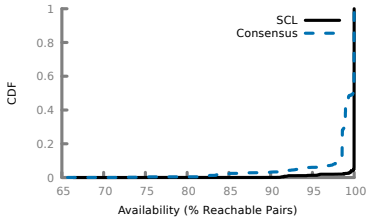


Figure 12: AS 1221: CDF of availability showing percentage of physically connected host pairs that are reachable based on routing rules.

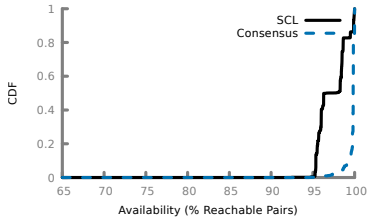


Figure 13: AS1239: CDF of availability showing percentage of physically connected host pairs that are reachable based on routing rules.

Topology	Switches	Links	Diameter	ND	CD
AS1221	83	236	0.63 s	0.77 s	0.72 s
AS1239	361	1584	0.54 s	0.67 s	0.54 s
Fat Tree	320	3072	2.52 ms	3.78 ms	1.336 ms

Table 1: Properties of the topologies for which we run evaluation.

the desired properties or invariants. For example, if we ignore the presence of partitions (where consensus mechanisms will always perform poorly), then we can ask how quickly a design responds to network events. We can express these delays using the notation that: $lat(x,y)$ is the network latency between points x and y; network nodes c are controllers and network nodes s are switches; and CD is the delay from waiting for the consensus algorithm to converge.

For SCL, the total response delay for an event detected by switch s is given by the sum of: (a) the time it takes for the event notification to reach a controller ($lat(s,c)$ for some c) and (b) the time for controller updates to reach all affected switches s' ($lat(c,s')$). Thus, we can express the worst case delay, which we call ND, as:

$$ND = \max_{s} \max_{s'} \min_{c} [lat(s,c)+lat(c,s')]$$

which represents the longest it can take for a switch s' to hear about an event at switch s, maximized over all s,s'.

When consensus-based approaches are used we must add the consensus time CD to ND (and in fact ND is an underestimate of the delay, because the delay is $[lat(s,c)+lat(c,s')]$ for a given c that is the leader, not the controller that minimizes the sum).[7]

At this simple conceptual level, three things become apparent: (i) typically SCL will respond faster than consensus-based systems because they both incur delay ND and only consensus-based systems incur CD, (ii) as the number of controllers increases, the relative performance gap between SCL and consensus-based approaches will increase (CD typically increases with more participants, while ND will decrease), and (iii) SCL has better availability (SCL only requires one controller to be up, while consensus-based designs require at least half to be up).

[7]But if the consensus protocol is used for leader election, then ND is actually an underestimate of the worst-case delay, as there is no minimization over c.

We further explore the first of these conclusions using simulations. In our simulations we compare SCL to an idealized consensus algorithm where reaching consensus requires that a majority of controllers receive and acknowledge a message, and these messages and acknowledgments do not experience any queuing delay. We therefore set CD to be the median round-trip time between controllers. We run our simulation on three topologies: two AS topologies (AS 1221 and 1239) as measured by the RocketFuel project [27], and a datacenter topology (a 16-ary fat-tree). We list sizes and other properties for these topologies in Table 1. Our control application computes and installs shortest path routes.

Single link failure We first measure time taken by the network to converge back to shortest path routes after a link failure. Note that here we are not measuring how long it takes for all controllers and switches to have a consistent view of the world (we return to that later), just when are shortest routes installed after a failure. For each topology, we measured the convergence time by failing random links and plot CDFs in Figure 5, 6, 7. These results show that SCL clearly restores routes faster than even an idealized Paxos approach, and typically faster than the worst-case bound ND (which would require the failure to be pessimally placed to achieve).

But one can also ask how long it takes to actually achieve full agreement, where one might think consensus-based algorithms would fare better. By full agreement we mean: (a) the dataplane has converged so that packets are routed along the shortest path, (b) all controllers have received notification for the link failure event and (c) every controller's logical configuration corresponds with the physical configuration installed in the data plane. For each topology, we measured convergence time by failing random links and plot CDFs in Figure 8, 9, 10. Thus, even when measuring when the network reaches full agreement, SCL significantly outperforms idealized Paxos. Note that what consensus-based algorithms give you is knowing *when* the controllers agree with each other, but it does not optimize how quickly they reach agreement.

Continuous link failure and recovery To explore more general failure scenarios, we simulate five hours with an ongoing process of random link failure and

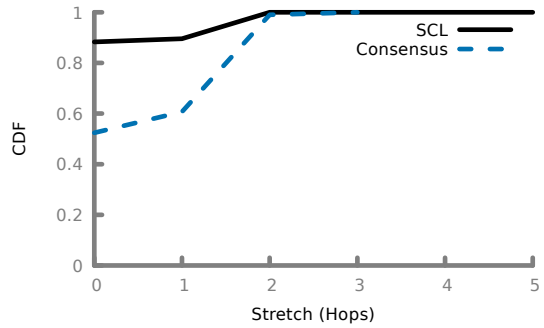


Figure 14: AS 1221: CDF of path inflation

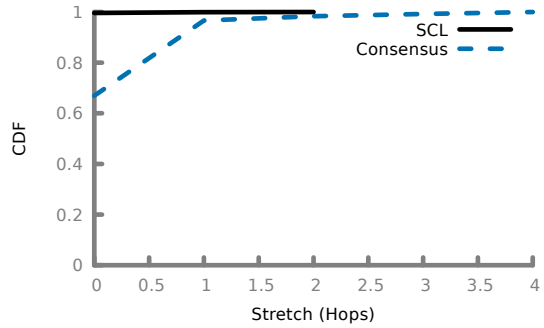


Figure 15: AS1239: CDF of path inflation.

recovery. The time between link failures in the network (MTBF) is 30 seconds, and each link's mean-time-to-recovery (MTTR) is 15 seconds; these numbers represent an extremely aggressive failure scenario. We look at two metrics: connectivity and stretch.

For connectivity, we plot the percentage of available[8] host pairs in Figures 11,12, 13. SCL and Paxos offer comparable availability in this case, but largely because the overall connectivity is so high. Note, that in Figure 13, the use of consensus does better than SCL– this is because in this particular simulation we saw instances of rapid link failure and recovery, and damping (*i.e.,* delaying) control plane responses paradoxically improves connectivity here. This has been observed in other context *e.g.,* BGP.

Even when hosts are connected, the paths between them might be suboptimal. To investigate this we plotted the stretch of all connected host-pairs (comparing the shortest path to the current path given by routing), as shown in Figures 14, 15. We did not observe any stretch in the datacenter topology, this can be explained by the large number of equal cost redundant paths in these topologies. As one can see, SCL significantly outperforms ideal Paxos in terms of path stretch in the other two topologies. In the case of AS1239, we can see that while SCL's rapid response and lack of damping affect availability slightly, the paths chosen by SCL are qualitatively better.

[8]By which we mean host pairs that are physically connected and can communicate using the current physical configuration

Traffic Type	Fat Tree	AS 1221	AS1239
Overall	63.428Kbps	205.828 Kbps	70.2 Kbps
Gossip	46.592 Kbps	3.2 Kbps	28.6 Kbps
Link Probing	16.329 Kbps	3.88 Kbps	36.4 Kbps
Routing Table Probing	0.0Kbps	248.4 Kbps	4.8Kbps

Table 2: Control Bandwidth usage for different topologies

7.2 Impact of Controller Disagreement

While the previous convergence results clearly indicate that SCL responds faster than consensus-based algorithms, one might still worry that in SCL the controllers may be giving switches conflicting information. We first observe that such an update race can happen only under rare conditions: controllers in SCL only update physical configuration in response to learning about new network events. If a single link event occurs, each controller would proactively contact the switch only after receiving this event and would install consistent configuration. A race is therefore only observable when two network events occur so close together in time so that different controllers observe them in different order, which requires that the time difference between these network events is smaller than the diameter of the network. But even in this case, as soon as all controllers have heard about each event, they will install consistent information. So the only time inconsistent information may be installed is during the normal convergence process for each event, and during such periods we would not have expected all switches to have the same configuration.

Thus, the impact of controller disagreement is minimal. However, we can more generally ask how often do controllers have an incorrect view of the network. In Figures 16, 17, 18 we show, that under the continuous failure/recover scenario described above, the CDF of how many links are incorrectly seen by at least one controller (where we periodically sample both the actual state of the physical network and the network state as seen by each controller). As we can see, typically there are very few disagreements between controllers and the physical state, and that SCL outperforms ideal Paxos in this regard. We also found that event logs between controllers agreed 99.4% of the time.

7.3 Bandwidth Usage

SCL uses broadcast for all control plane communication. This has several advantages: it avoids the need for bootstrapping (*i.e.,* running a routing algorithm on the control plane, which then allows the controllers to manage the data plane), is extremely robust to failures, and provides automatic alignment between data plane and control plane connectivity. This last one is particularly important, because otherwise one would have to handle special cases such as where controllers within a given data plane partition might not be able to talk to each other. Such cases would unavoidably complicate the controller logic,

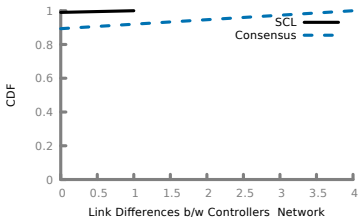


Figure 16: CDF of number of times network state disagreed with controller's network state on a fat tree.

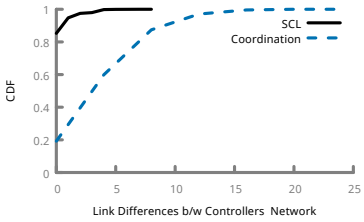


Figure 17: AS 1221: CDF of number of times network state disagreed with any controller's network state.

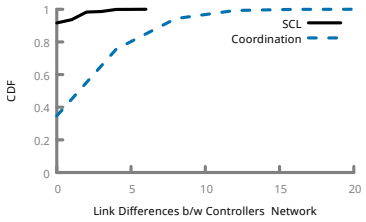


Figure 18: AS1239: CDF of number of times network state disagreed with any controller's network state.

System	First Update			Correct		
	Median	Min	Max	Median	Min	Max
SCL	117.5ms	91ms	164ms	268.5ms	201ms	322ms
ONOS	159.5ms	153ms	203ms	276.5ms	164ms	526ms

Table 3: Time taken before a controller begins reacting to a network event (First Update) and before the network converges (Correct).

and make eventual correctness harder to achieve.

However, one might think that this requires too much bandwidth to be practical. Here we consider this question by running a simulation where the mean time between link failures (MTBF) is 10 minutes, and the mean time for each link to recover is 5 minutes. The control traffic depends on the control plane parameters, and here we set SCL's gossip timer at 1 second and network state query period at 20 seconds. Table 2 shows the control plane's bandwidth consumption for the topologies we tested. On average we find that SCL uses a modest amount of bandwidth. We also looked at instantaneous bandwidth usage (which we measured in 0.5 second buckets) and found that most of the time peak bandwidth usage is low: for the fat tree topology 99.9% of the time bandwidth is under 1Mbps, for AS1221 98% of the time bandwidth is under 1Mbps, and for AS1239 97% of the time bandwidth is under 1Mbps.

One might ask how the bandwidth used on a given link scales with increasing network size. Note that the dominant cause of bandwidth usage is from responses to link and routing table probing messages. The number of these messages grows as the number of switches (not the number of controllers), and even if we increase the number of switches three orders of magnitude (resulting in a network with about 100K switches, which is larger than any network we are aware of) the worst of our bandwidth numbers would be on the order of 200Mbps, which is using only 2% of the links if we assume 10Gbps links. Furthermore, the failure rate is extremely high; we would expect in a more realistic network setting that SCL could scale to even large networks with minimal bandwidth overhead.

7.4 Response Time of the SCL Implementation

The primary metric by which we compare our implementation to existing distributed controller frameworks is response time. We begin by showing the improvements achieved by our implementation of SCL when compared to a traditional distributed controller in a datacenter network. We ran this evaluation on CloudLab [1], where the dataplane network consisted of 20 switches connected in a fat tree topology (using 48 links). Each of the dataplane switches ran OpenVSwitch (version 2.5.0). In the control plane we used *three* replicated controllers. To fit into the CloudLab environment we modified SCL to use an out-of-band control channel: each controller-proxy established a TCP connection with all switch-agents and forwarded any network events received from the switch on all of these connections. For comparison we used ONOS (version 1.6.0). Both controllers were configured to compute shortest path routes.

In our experiments we measured the time taken for paths to converge to their correct value after a single link failure in the network. We measured both the time before a controller reacts to the network event (First Update in the table) and before all rule updates are installed (Correct). We repeated this test 10 times and report times in milliseconds elapsed since link failures (Table 3). We find that SCL consistently responds to network events before ONOS; this is in line with the fact that ONOS must first communicate with at least one other controller before installing rules. In our experiments, we found that this could induce a latency of up to 76ms before the controller issues its first updates. In the median case, SCL also achieved correctness before ONOS, however the gap is smaller in this case. Note, however, the latencies in this setup are very small, so we expected to find a small performance gap.

8 Conclusion

Common wisdom holds that replicated state machines and consensus are key to implementing distributed SDN controllers. SCL shows that SDN controllers eschewing consensus are both achievable and desirable.

9 Acknowledgment

We thank Colin Scott, Amin Tootoonchian, and Barath Raghavan for the many discussions that shaped the ideas in this paper; we thank Shivaram Venkataraman, our shepherds Minlan Yu and Jay Lorch, and the anonymous reviewers for their helpful comments. This work was funded in part by a grant from Intel Corporation, and by NSF awards 1420064 and 1616774.

References

[1] A. Akella. Experimenting with Next-Generation Cloud Architectures Using CloudLab. *IEEE Internet Computing*, 19:77–81, 2015.

[2] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, et al. ONOS: Towards an Open, Distributed SDN OS. In *HotSDN*, 2014.

[3] A. Elwalid, C. Jin, S. H. Low, and I. Widjaja. MATE: MPLS Adaptive Traffic Engineering. In *INFOCOM*, 2001.

[4] S. Ghorbani and M. Caesar. Walk the Line: Consistent Network Updates with Bandwidth Guarantees. In *HotSDN*, 2012.

[5] J. Gray. Notes on Data Base Operating Systems. In *Advanced Course: Operating Systems*, 1978.

[6] H. Howard. ARC: Analysis of Raft Consensus. *Technical Report UCAM-CL-TR-857*, 2014.

[7] H. Howard, D. Malkhi, and A. Spiegelman. Flexible Paxos: Quorum intersection revisited. *CoRR*, abs/1608.06696, 2016.

[8] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, 2013.

[9] S. Kandula, D. Katabi, B. S. Davie, and A. Charny. Walking the Tightrope: Responsive Yet Stable Traffic Engineering. In *SIGCOMM*, 2005.

[10] N. Katta, H. Zhang, M. Freedman, and J. Rexford. Ravana: Controller fault-tolerance in software-defined networking. In *SOSR*, 2015.

[11] N. P. Katta, J. Rexford, and D. Walker. Incremental Consistent Updates. In *HotSDN*, 2013.

[12] D. Katz and D. Ward. Bidirectional Forwarding Detection (BFD). RFC 5880 (Proposed Standard), June 2010. Updated by RFCs 7419, 7880.

[13] D. Katz and D. Ward. Bidirectional Forwarding Detection (BFD) for IPv4 and IPv6 (Single Hop). RFC 5881 (Proposed Standard), June 2010.

[14] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.

[15] L. Lamport, R. E. Shostak, and M. C. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, 1982.

[16] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[17] R. Mahajan and R. Wattenhofer. On consistent updates in Software Defined Networks. In *HotNets*, 2013.

[18] J. Mccauley. POX: A Python-based OpenFlow Controller. http://www.noxrepo.org/pox/about-pox/.

[19] J. McCauley, A. Panda, M. Casado, T. Koponen, and S. Shenker. Extending SDN to Large-Scale Networks. In *Open Networking Summit*, 2013.

[20] J. McClurg, N. Foster, and P. Cerný. Efficient Synthesis of Network Updates. *CoRR*, abs/1403.5843, 2015.

[21] T. Mizrahi, E. Saat, and Y. Moses. Timed consistent network updates. In *SOSR*, 2015.

[22] D. Ongaro. Bug in Single-Server Membership Changes. raft-dev post 07/09/2015, https://goo.gl/a7hKXb, 2015.

[23] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker. CAP for Networks. In *HotSDN*, 2013.

[24] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *NSDI*, 2015.

[25] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *SIGCOMM*, 2012.

[26] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. consistent Updates for Software-Defined Networks: Change You Can Believe In! In *HotNets*, 2011.

[27] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *SIGCOMM*, 2002.

[28] A. Tootoonchian and Y. Ganjali. HyperFlow: A distributed control plane for OpenFlow. In *INM/WREN*, 2010.

A Safety Policies in SCL

Next we look at how SCL ensures that *safety policies* always hold. From our definitions, safety policies must never be violated, even in the presence of an arbitrary sequence of network events. Unlike liveness policies, we cannot rely on the control plane to enforce these policies. We therefore turn to data plane mechanisms for enforcing these policies.

As noted earlier, safety policies cannot have any dependence on what other paths are available in the network (disallowing policies affecting optimality, *e.g.,* shortest path routing) or on what other traffic is in the network (disallowing policies which ensure traffic isolation). The safety policies listed in §3.2 meet this requirement: waypointing requires that all chosen paths include the waypoint, while isolation limits the set of endpoints that valid paths can have. Next, we present our mechanism for enforcing safety policies, and then provide an analysis showing that this mechanism is both necessary and sufficient.

A.1 Mechanism

Safety policies in SCL constrain the set of valid paths in the network. We assume the control applications only generate paths that adhere to these constraints. Such policies can therefore be implemented by using dataplane consistency mechanisms to ensure that packets only follow paths generated by a control application.

Our mechanisms for doing this extends existing work on consistent updates [4, 25]. However in contrast to these works (which implicitly focus on planned updates), our focus is explicitly restricted to unplanned topology updates. The primary aim of this mechanism is to ensure that each packet follows exactly one controller generated path. Similar to prior work, we accomplish this by associating a label with each path, and tagging packets on ingress with the label for a given path. Packets are then routed according to this label, ensuring that they follow a single path. Packets can be tagged by changing the VLAN tag, adding an MPLS label, or using other packet fields.

In SCL, the controller-proxy is responsible for ensuring that packets follow a single path. Our mechanism for doing this is based on the following observation: since we assume controllers are *deterministic*, the path (for a single packet) is determined entirely by the current network state and policy. The controller-proxy therefore uses a hash of the network state and policy as a policy label (Π). When a controller sends its controller-proxy a flow table update, the controller-proxy modifies each match predicate in the update. The match predicates are updated so that a packet matches the new predicate if and only if the packet is tagged with Π and if it would have matched the old predicate (*i.e.,* given a match-action rule r with match m, the controller-proxy produces a rule r' which will only match packets which are tagged with Π and match m). The controller-proxy also augments the update to add rules which would tag packets on ingress with an appropriate tag. The controller-proxy then sends these rules to the appropriate switch (and its switch-agent). Once these rules are installed, packets are forwarded along exactly one path from ingress to egress, and are dropped when this is not possible.

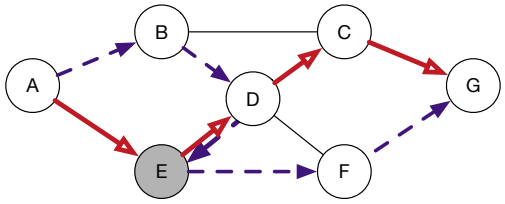


Figure 19: Example where waypointing is violated when using inconsistent paths.

Since the labels used by SCL are based on the network state and policy, controllers in agreement will use the same label, and in steady state all controllers will install exactly one set of rules. However, during periods of disagreement, several sets of rules might be installed in the network. While these do not result in a correctness violation, they can result in resource exhaustion (as flow table space is consumed). Garbage collection of rules is a concern for all consistent update mechanisms, and we find that we can apply any of the existing solutions [11] to our approach. Finally, we observe that this mechanism need not be used for paths which do not enforce a safety policy. We therefore allow the controller to mark rules as not being safety critical, and we do not modify the match field for such rules, reducing overhead. We do not require that single-image controllers mark packets as such, this mechanism merely provides an opportunity for optimization when control applications are SCL aware.

A.2 Analysis

Next we show that this mechanism is both necessary and sufficient for ensuring that safety policies hold. Since we assume that controllers compute paths that enforce safety policies, ensuring that packets are only forwarded along a computed path is *sufficient* to ensure that safety policies are upheld. We show necessity by means of a counterexample. We show that a packet that starts out following one policy compliant path, but switches over partway to another policy compliant path can violate a safety policy. Consider the network in Figure 19, where all packets from A to G need to be waypointed through the shaded gray node E. In this case, both the solid-red path and the dashed purple path individually meet this requirement. However, the paths intersect at both D and E. Consider a case where routing is done solely on packet headers (so we do not consider input ports, labels, etc.). In this case a packet can follow the path A—B—D—C—G, which is a combination of two policy compliant paths, but is not itself policy compliant. Therefore, it is *necessary* that packets follow

a single path to ensure safety policies are held. Therefore our mechanism is both *necessary* and *sufficient*.

A.3 Causal Dependencies

Safety policies in reactive networks might require that causal dependencies be enforced between paths. For example, reactive controllers can be used to implement stateful firewalls, where the first packet triggers the addition of flow rules that allow forwarding of subsequent packets belonging to the connection. Ensuring correcntess in this case requires ensuring that all packets in the connection (after the first) are handled by rules that are causally after the rules handling the first packet. Our mechanism above does not guarantee such causality (since we do not assume any ordering on the labels, or on the order in which different controllers become aware of network paths). We have not found any policies that can be implemented in proactive controllers (which we assume) that require handling causal dependencies, and hence we do not implement any mechanisms to deal with this problem. We sketch out a mechanism here using which SCL can be extended to deal with causal dependencies.

For this extension we require that each controller-proxy maintain a vector clock tracking updates it has received from each switch, and include this vector clock in each update sent to a switch-agent. We also require that each switch-agent remember the vector clock for the last accepted update. Causality can now be enforced by having each switch-agent reject any updates which happen-before the last accepted update, *i.e.,* on receiving an update the switch-agent compares the update's vector clock v_u with the vector-clock for the last accepted update v_a, and rejects any updates where $v_u \preceq v_a$. The challenge here is that the happens-before relation for vector clocks is a partial order, and in fact v_u and v_a may be incomparable using the happens-before relation. There are two options in this case: (a) the switch-agent could accept incomparable updates, and this can result in causality violations in some cases; or (b) the switch-agent can reject the incomparable update. The latter is safe (*i.e.,* causality is never violated), however it can render the network unavailable in the presence of partitions since controllers might never learn about events known only to other controllers across a partition.

B Policy Changes in SCL

All the mechanisms presented thus far rely on the assumption that controllers in the network agree about *policy*. The *policy coordinator* uses 2-phase commit [5] (2PC) to ensure that this holds. In SCL, network operators initiate policy changes by sending updates to the *policy coordinator*. The policy coordinator is configured with the set of active controllers, and is connected to each controller in this set through a reliable channel (*e.g.,* established using TCP). On receiving such an update, the policy coordinator uses 2PC to update the controllers as follows:

1. The policy coordinator informs each controller that a policy update has been initiated, and sends each controller the new policy.
2. On receiving the new policy, each controller sends an acknowledgment to the policy coordinator. Controllers also start start queuing network events (*i.e.,* do not respond to them) and do not respond to them until further messages are received.
3. Upon receiving an acknowledgement from all controllers, the policy coordinator sends a message to all controllers informing them that they should switch to the new policy.
4. On receiving the switch message, each controller starts using the new policy, and starts processing queued network events according to this policy.
5. If the policy coordinator does not receive acknowledgments from all controllers, it sends an abort message to all controllers. Controllers receiving an abort message stop queuing network events and process both queued events and new events according to the old policy.

Two phase commit is not live, and in the presence of failures, the system cannot make progress. For example, in the event of controller failure, new policies cannot be installed and any controller which has received a policy update message will stop responding to network events until it receives the switch message. However, we assume that policy changes are *rare* and performed during periods when an administrator is actively monitoring the system (and can thus respond to failures). We therefore assume that either the policy coordinator does not fail during a policy update or can be restored when it does fail, and that before starting a policy update network administrators ensure that all controllers are functioning and reachable. Furthermore, to ensure that controllers are not stalled from responding to network events forever, SCL controllers gossip about *commit* and *abort* messages from the 2PC process. Controllers can commit or abort policy updates upon receiving these messages from other controllers. This allows us to reduce our window of vulnerability to the case where either (a) the policy coordinator fails without sending any commits or aborts, or (b) a controller is partitioned from the policy coordinator and all other controllers in the system.

The first problem can be addressed by implementing fault tolerance for the policy coordinator by implementing it as a replicated state machine. This comes at the cost of additional complexity, and is orthogonal to our work.

The second problem, which results from a partition preventing the policy coordinator from communicating with a controller, cannot safely be solved without repairing the partition. This is not unique to SCL, and is true for

all distributed SDN controllers. However, in some cases, restoring connectivity between the policy coordinator and all controllers might not be feasible. In this case network operators can change the set of active controllers known to the policy agent. However it is essential that we ensure that controllers which are not in the active set cannot update dataplane configuration, since otherwise our convergence guarantees do not hold. Therefore, each SCL agent is configured with a set of *blacklisted* controllers, and drops any updates received from a controller in the blacklisted set. We assume that this blacklist can be updated through an out of band mechanism, and that operators blacklist any controller before removing them from the set of active controllers. Re-enabling a blacklisted controller is done in reverse, first it is added to the set of active controllers, this triggers the 2PC mechanism and ensures that all active controllers are aware of the current configuration, the operator then removes the controller from the blacklist.

Finally, note that SCL imposes stricter consistency requirements in responding to policy changes when compared to systems like ONOS and Onyx, which store policy using a replicated state machine; this is a trade-off introduced due to the lack of consistency assumed when handling network events. This is similar to recent work on consensus algorithms [7] which allow trade-offs in the number of nodes required during commits vs the number of nodes required during leader election.

B.1 Planned Topology Changes

Planned topology changes differ from unplanned ones in that operators are aware of these changes ahead of time and can mitigate their effects, *e.g.,* by draining traffic fron links that are about to be taken offline. We treat such changes as policy changes, *i.e.,* we require that operators change their policy to exclude such a link from being used (or include a previously excluded link), and implement them as above.

B.2 Load-Dependent Update

Load-dependent updates, which include policies like traffic engineering, are assumed by SCL to be relatively infrequent, occurring once every few minutes. This is the frequency of traffic engineering reported in networks such as B4 [8], which aggressively run traffic engineering. In this paper we focus exclusively on traffic engineering, but note that other load-dependent updates could be implemented similarly.

We assume that traffic engineering is implemented by choosing between multiple policy-compliant paths based on a traffic matrix, which measures demand between pairs of hosts in a network. Traffic engineering in SCL can be implemented through any of three mechanisms, each of which allows operators to choose a different point in the trade-off space: (a) one can use techniques like TeXCP [9] and MATE [3] which perform traffic engineering entirely in the data plane; or (b) operators can update traffic matrices using the policy update mechanisms.

Dataplane techniques including TeXCP can be implemented in SCL unmodified. In this case, control plane applications must produce and install multiple paths between hosts, and provide mechanisms for a switch or middlebox to choose between these paths. This is compatible will all the mechanisms we have presented thus far; the only additional requirement imposed by SCL is that the field used to tag paths for traffic engineering be different from the field used by SCL for ensuring dataplane consistency. These techniques however rely on local information to chose paths, and might not be sufficient in some cases.

When treated as policy changes, each traffic matrix is submitted to a *policy coordinator* which uses 2PC to commit the parameters to the controller. In this case, we allow each update process to use a different coordinator, thus providing a trivial mechanism for handling failures in policy coordinators. However, similar to the policy update case, this mechanism does not allow load-dependent updates in the presence of network partitions.

Robust validation of network designs under uncertain demands and failures

Yiyang Chang, Sanjay Rao and Mohit Tawarmalani
Purdue University

Abstract

A key challenge confronting wide-area network architects is validating that their network designs provide assurable performance in the face of variable traffic demands and failures. Validation is hard because of the exponential, and possibly non-enumerable, set of scenarios that must be considered. Current theoretical tools provide overly conservative bounds on network performance since to remain tractable, they do not adequately model the flexible routing strategies that networks employ in practice to adapt to failures and changing traffic demands. In this paper, we develop an optimization-theoretic framework to derive the worst-case network performance across scenarios of interest by modeling flexible routing adaptation strategies. We present an approach to tackling the resulting intractable problems, which can achieve tighter bounds on network performance than current techniques. While our framework is general, we focus on bounding worst-case link utilizations, and case studies involving topology design, and MPLS tunnels, chosen both for their practical importance and to illustrate key aspects of our framework. Evaluations over real network topologies and traffic data show the promise of the approach.

1 Introduction

In designing wide-area networks for ISPs and cloud service providers, it is critical to ensure predictable performance at acceptable costs. However, achieving this goal is challenging because links fail (both owing to planned maintenance, and unplanned events such as fiber cuts and equipment failures) [38, 50, 23], and network traffic is variable [9] and constantly evolving [23].

Validating that a network can cope with a range of traffic conditions and failure scenarios is challenging because the number of scenarios to consider are typically exponentially many, and may even be non-enumerable. For instance, a common requirement is to verify that a network with N links can service demand for all combinations of f simultaneous link failures [50, 48, 37]. The number of failure scenarios to consider is $\binom{N}{f}$ for each demand. Further, the set of traffic matrices are not even enumerable, so naively considering all traffic matrices and failure scenarios is prohibitive.

There is a huge gap between practice and existing theoretical tools. Oblivious routing [40, 41, 9, 49], and more generally, robust optimization [12, 14] allow bounding worst-case performance across multiple scenarios of interest. However, to ensure tractability of the problem, these techniques make the *conservative* assumption that the network cannot adapt to changes in demands by re-routing traffic [40, 41, 9, 49], or admit limited forms of adaptation [50, 15]. In practice, networks do adapt by re-routing traffic as demands shift or failures occur, and such adaptation can make network operations much more efficient. Further, the advent of Software-Defined Networking (SDN) allows for network-wide optimization, and facilitates the deployment of flexible re-routing strategies [30, 31].

Given the large gap between theory and practice, the process of validating network designs today is ad-hoc, often requiring extensive simulations, which can be highly time consuming as well as fall short of guaranteeing provable bounds on network performance. In this paper, we take a first step towards tackling this by presenting a formal framework to provide performance bounds on a network design across a set of scenarios (demands, failures). The key novelty in our framework is that it can accommodate a richer set of adaptation mechanisms, used in practice today, for re-routing traffic on failures and changes in demands.

When flexible routing strategies are considered, providing robust performance guarantees typically requires solving intractable non-convex (and often non-linear) optimization problems. We address these difficulties by leveraging cutting-edge techniques in the non-linear optimization literature [44]. An attractive aspect of these techniques is their generality, which allows them to be

applied to a wide range of network validation problems. We show that these techniques lead to tighter bounds on the validation problem than existing state-of-the-art approaches in robust optimization, a finding that has applications beyond networking. Further. the bounds are tight in practical settings of interest - e.g., when demands are expressed as a convex combination of known historical demands [49]. Finally, we show how the techniques may be augmented with analysis of individual problem structure to substantially improve the quality of bounds.

For concreteness, we focus on link utilization, a widely accepted traffic engineering metric [9, 49, 50], which impacts application latency and throughput. We apply our framework to two contrasting, yet practical case studies to illustrate key aspects of our framework. The case studies differ in the type of uncertainty (failures and demands), and the type of adaptation. Specifically, we consider (i) multi-commodity flow (MCF) routing [21, 9, 50] which provides the most flexibility and efficiency, and (ii) MPLS-style tunneling [30, 29] which has more limited flexibility in routing.

While we focus on validation, our framework can enable the synthesis of designs with performance guarantees under uncertainty. We demonstrate this by showing how our approach can aid operators in determining the most effective ways to augment link capacities while ensuring acceptable link utilizations under failures.

We evaluate our approach using multiple real topologies [6] and public traffic data [1]. Our framework performs better than oblivious formulations for both case studies, while surprisingly matching optimal in all the experiments for the failure case study. Further, we show our framework aids in (i) identifying bad failure scenarios; (ii) determining how to best augment link capacity to handle failures; and (iii) evaluating design heuristics – e.g., we show the potential for poor performance with common tunnel selection heuristics.

2 Motivation

2.1 Robust validation applications

A network design consists of (i) *invariant parameters,* which cannot be changed (or are costly to change) across failures and/or demands; and (ii) *adaptable parameters,* which may be flexibly chosen for any scenario. Our framework ensures that the choice of invariant parameters is acceptable across a set of demands and/or failures. Below, we present motivating examples.

Topology Design. In designing network topologies, operators must determine what links to lease and how much capacity to provision. While the set of links and their capacities is difficult to change across failures and demands, the network may adapt by re-routing traffic.

MPLS Tunnel Selection. A common traffic engineering practice is to use tunnels (e.g., MPLS [42]) between each ingress and egress switch, to ensure a core network that does not need to run the BGP protocol. In such settings, a light-weight adaptation mechanism is to switch traffic across k pre-selected tunnels between each source destination pair, which only involves changing flow tables in appropriate ingress switches [29, 30]. Changing tunnels is more heavy-weight since the flow tables of internal switches also need to be modified. A good choice of pre-selected tunnels can lower the frequency of changing tunnels in response to fluctuations in demand.

Middlebox placement. Network policy may require that some of the flows traverse a set of middleboxes such as firewalls and intrusion detection systems (IDS) [39, 7]. While the placement of network middleboxes typically occurs over relatively longer time-scales, traffic may be re-routed to handle normal traffic fluctuations or failures.

In these examples, the topology itself, and the set of tunnels and placement of middleboxes as applicable are *invariant* parameters, while the fraction of traffic sent along a given tunnel is an *adaptable* parameter.

Robust validation may be performed at initial design time, as well as in a *continual* fashion as the network evolves, and new projections on demands are available. Robust validation may indicate the network is no longer able to cope with the scenarios of interest, requiring the operator to consider changes to the design (e.g., by provisioning more capacity on links). Further, it can provide information on which scenario causes the network requirements to be violated, and aid in determining design changes to address the violations.

2.2 Robust validation framework

Our framework is closely related to robust optimization. In traditional robust optimization, input parameters belong to an uncertainty set, and the objective is minimized across any parameter choice in the set [12, 14]. Further, recourse actions may be considered that depend on the specific parameter value. In the networking context, a typical recourse action involves rerouting traffic to handle a change in traffic matrix or failure. The robust optimization literature considers limited forms of recourse actions, primarily for tractability reasons, which may lead to more conservative estimates of worst-case performance (§4.4). In contrast, we model richer network adaptation, and tackle the resulting intractable problems. Prior approaches can be seen as special cases of our more general framework discussed below:

Metrics to capture performance of network design. Our framework can validate a variety of network metrics such as link utilizations, and bandwidth assigned to latency sensitive flows. For concreteness, in this paper, we focus on the utilization of the most congested link (which we will refer to as *Maximum Link Utilization (MLU)*, a widely used objective function [9, 49, 50]. Though we do

not discuss this extensively, our framework also applies to other common metrics of link utilizations (e.g., sum of penalties assigned to individual links, where penalties are convex functions of link utilizations [48, 22, 24]). We focus on utilizations given their extensive use in the traffic engineering literature, and since they reflect application performance (e.g., throughput for bandwidth sensitive applications is inversely related to utilizations).

Characterizing uncertainty in network conditions. We seek to validate that a network design performs well across demands and failure scenarios of interest. A typical set of failure scenarios to consider is all simultaneous failures of F or fewer links [50, 48]. The range of demands may be specified in multiple ways. A common model is to specify a set of historical traffic matrices, and require that all demands based on standard prediction models are considered. We formally discuss this model as well as other models in §4.3 and §5.2.

Modeling how networks adapt. Networks may respond to failures, and changes in demand by rerouting traffic in the best possible fashion to keep utilizations low. This can be achieved by determining the optimal routing (MCF) for a given scenario. This design point is becoming increasingly practical with the adoption of SDNs, given that periodic reoptimization for network state is feasible. Other models may allow adaptation, but with constraints. For instance, in the MPLS tunneling example, the network may adapt by changing how traffic is split across pre-selected tunnels between each ingress and egress pair, though the tunnels themselves do not change. This corresponds well to SDN deployments where only edge routers are SDN enabled [17]. Finally, policy constraints (e.g., a requirement that a set of middleboxes be traversed) may constrain how networks may adapt [47, 39, 7].

3 Formalizing robust validation

3.1 General problem structure

Let X denote the uncertainty set (possibly continuous and non-enumerable) of demands, or failures over which a given network design must be validated. The design includes all parameters that must remain invariant with changes in demands and failures (e.g., network topology, selection of tunnels, placement of middleboxes). For any given scenario $x \in X$, the network may adapt by routing traffic appropriately as described in §2.2.

Let y denote the parameters determined by the network when adapting to scenario x. This includes how traffic is routed – e.g., in the tunneling context, y includes parameters that capture how traffic must be split across tunnels – though there may be additional variables determined as we discuss in §3.2. Formally, the network validation problem may be written as:

$$F^* = \max_{x \in X} \min_{y \in Y(x)} F(x,y) \quad (1)$$

The *inner* minimization captures that for any given scenario $x \in X$, the network determines y in a manner that minimizes an objective function $F(x,y)$ from a set of permissible strategies $Y(x)$. For the fully flexible routing model, $Y(x)$ corresponds to strategies permitted by the standard MCF constraints [21], while for routing with middlebox policies, only strategies that ensure the desired set of middleboxes are traversed are permitted. The *outer* maximization robustly captures the worst-case performance across the set of scenarios X, assuming the network adapts in the best possible fashion for each x.

In this paper, we focus on objective functions $F(x,y)$ that minimize the MLU as discussed in §2.2. We refer to (1) as the validation problem, since it can be used to verify that a chosen design meets a desired utilization goal. For instance, when applied to topology design, $F^* > 1$ indicates the network is not sufficiently provisioned to handle all failures and demands of interest.

For any given scenario x, the inner problem is typically easy to solve (a linear program (LP)), since the network must compute y online to adapt to any failure or shift in demand. The validation problem is however challenging since exponentially many (and potentially non-enumerable) scenarios x must be considered.

3.2 Concrete validation problems

We next relate the general formulation (1) to two concrete case studies, chosen both for their practical importance and to illustrate key ideas of the framework.

- The first case study validates topology design against failures, with the most flexible network adaptation.
- The second example validates tunnel selection across variable demands, with network adaptivity constrained to splitting traffic across pre-selected tunnels.

The examples illustrate the generality of our framework in terms of its ability to consider both failures and demands (discrete and continuous uncertainty sets), and different types of adaptivity models (flexible and more constrained). However, our framework applies to a wider range of applications including simultaneously varying demands and failures, other adaptation models such as middlebox constraints, and other ways of combining adaptation models and uncertainty sets (§5).

We use the notation $x = (x^f, x^d)$ where x^f denotes a failure scenario and x^d denotes a particular demand, dropping superscripts when the context is clear. Likewise, we use $y = (r, U)$ where r denotes how traffic is routed, and U denotes utilization metrics computed as a result. Since our focus is on minimizing MLU, the inner problem may be expressed as $\min_{y \in Y(x)} U$, with constraints in $Y(x)$ which express the requirement that the

$$Y(x) = \left\{ (r,U) \;\middle|\; \begin{array}{ll} \gamma_k(x)U \geq \sum_{i\in I} \beta_{ik}(x) r_i & k \in K \\ \sum_{i\in I} \alpha_{ij}(x) r_i \geq \delta_j(x) & j \in J \\ r \geq 0 & \end{array} \right\}$$
$$F(x,r,U) = U$$

Figure 1: General structure of determining routes (r) for scenario x to minimize MLU (U).

$$\begin{array}{lll} (W) \max\limits_{x,v,\lambda} & \sum_{j\in J} \delta_j(x) v_j & \\ \text{s.t.} & \sum_{j\in J} \alpha_{ij}(x) v_j \leq \sum_{k\in K} \beta_{ik}(x) \lambda_k & i \in I \\ & \sum_{k\in K} \gamma_k(x) \lambda_k = 1 & \\ & x \in X, (v_j)_{j\in J} \geq 0, (\lambda_k)_{k\in K} \geq 0 & \end{array}$$

Figure 2: General structure of validation problem derived from Figure 1 as a single-stage formulation.

utilization of every link is at most U. We now discuss how constraints $Y(x)$ are specified for our case studies.

Fully flexible routing under uncertain failures. Let x^f_{ij} be a binary variable which is 1 if link $\langle i,j\rangle \in E$ (the set of links) has failed, and 0 otherwise. Since we do not consider variable demands in this case study, we let d_{it} denote the known demand from source i to destination t. Let r_{ijt} denote the total traffic to t carried on link $\langle i,j\rangle$. Let c_{ij} denote the capacity of link $\langle i,j\rangle$. Then, $Y(x)$ corresponds to the standard MCF constraints [21], and may be expressed as:

$$\begin{array}{l} U c_{ij}(1 - x^f_{ij}) \geq \sum_t r_{ijt} \quad \langle i,j\rangle \in E \\ \sum_j r_{ijt} - \sum_j r_{jit} = \begin{cases} d_{it} & \forall t, i \neq t \\ -\sum_j d_{jt} & \forall t, i = t \end{cases} \\ r_{ijt} \geq 0 \quad \forall i,j,t \end{array} \tag{2}$$

The first constraint ensures that (i) the utilization of link $\langle i,j\rangle$ is at most U for all non-failed links; and (ii) no traffic is carried on a failed link. The second constraint captures flow balance requirements. Specifically, the net outflow from node i to destination t is the total traffic destined to t when $i = t$, and d_{it} otherwise.

Tunnel constraints and uncertain demands. Given a set of pre-selected tunnels, let T_{ijstk} be a binary parameter that denotes whether the link $\langle i,j\rangle$ is on tunnel k for traffic from the source s to destination t. Let x^d_{st} denote the total $s-t$ traffic, and r_{stk} the subset of this traffic on tunnel k. Then, $Y(x)$ may be expressed as:

$$\begin{array}{l} U c_{ij} \geq \sum_{s,t,k} r_{stk} T_{ijstk} \quad \langle i,j\rangle \in E \\ \sum_k r_{stk} = x^d_{st} \quad \forall s,t; \quad r_{stk} \geq 0 \quad \forall s,t,k \end{array} \tag{3}$$

The first constraint ensures that the utilization of every link is bounded by U. The second constraint captures that the sum of the traffic on all tunnels k for each $s-t$ pair must add up to the total demand of that pair.

3.3 Non-linear reformulation

The validation problem in (1) has been represented in a form referred to as a *two-stage formulation* (e.g., [15]). In the two-stage problem, the optimal second-stage variables (y) depend on the first-stage (x). We simplify this problem by re-expressing it as a single-stage problem, where all the variables are determined simultaneously.

In many network validation problems, including our case studies, the inner problem $\min_{y\in Y(x)} F(x,y)$ is an LP in variable $y = (r,U)$ for a fixed scenario x. This is reasonable because online adaptations of y must be computationally efficient. Figure 1 shows the general structure of the LP. Notice that the coefficients depend on scenario x. For example, in the failure validation case study, $\alpha_{ij}(x)$, $\beta_{ik}(x)$, and $\delta_j(x)$ are constants while $\gamma_k(x)$ is a linear function of x. For a specific value of x, the inner problem is an LP.

It is well known that every LP (referred to as a primal form) involving a minimization objective may be converted into an equivalent maximization LP (referred to as a dual form) which achieves the same objective (assuming the dual is feasible) [19]. The validation problem can then be expressed as a single-stage formulation by:

1. Rewriting $\min_{y\in Y(x)} F(x,y)$ as an equivalent maximization problem using LP duality.
2. Adding the constraints $x \in X$ to the dual form to capture the set of demands or failure scenarios of interest.

Figure 2 shows the general structure of the validation problem as a single-stage formulation. Notice that variables r and U in Figure 1 have been replaced by the dual variables λ and v. Moreover, x is now a variable since the problem validates utilization over all uncertain scenarios.

Formulations **(F)** and **(V)** in Figure 3 capture the validation problem for our case studies involving failures (2) and variable demands (3) respectively. At first glance, both formulations appear non-linear – the objective in (V) involves products of x^d and u variables, while the second constraint of (F) involves a product of variables x^f_{ij} and λ_{ij}. In §4.2, we show that (F) can be written as an integer program (IP) when X is the set of scenarios involving the failure of f or fewer links simultaneously. Regardless, both (V) and (F) are hard problems (non-linear non-convex and IP respectively).

4 Making validation tractable

§3.3 has shown that the validation problems, including our case studies, are typically intractable, Given the intractable nature of the problems, we do not solve them to optimality, rather seek ways to obtain upper bounds on the true optimal of (1). Since the purpose of validation is to ensure a design is acceptable, an upper bound that satisfies the design criteria is sufficient.

We aim for a general approach to tackle a wide range

$$(F)\max_{v,\lambda,x} \quad \sum_{t,i\neq t} d_{it}(v_{it}-v_{tt})$$
$$\text{s.t.} \quad v_{it}-v_{jt}\leq \lambda_{ij} \quad \forall t,\langle i,j\rangle\in E$$
$$\sum_{\langle i,j\rangle\in E}\lambda_{ij}c_{ij}(1-x^f_{ij})=1$$
$$x^f\in X;\quad x^f_{ij}\in\{0,1\};\quad \lambda_{ij}\geq 0,\quad \langle i,j\rangle\in E$$

$$(V)\max_{v,\lambda,x} \quad \sum_{s,t} x^d_{st}v_{st}$$
$$\text{s.t.} \quad v_{st}\leq \sum_{\langle i,j\rangle\in E} T_{ijstk}\lambda_{ij} \quad \forall s,t,k$$
$$\sum_{\langle i,j\rangle\in E}\lambda_{ij}c_{ij}=1$$
$$x^d\in X;\quad \lambda_{ij}\geq 0,\quad \langle i,j\rangle\in E$$

Figure 3: Formulations of validation problems for failure case study (F), and tunnel selection case study (V).

of validation problems. In the optimization literature, problems such as (1) are referred to as robust optimization problems and have been tackled mostly for limited adaptations. Instead, we use non-linear programming techniques, and show they achieve better bounds, a finding that has applications beyond networking.

We introduce the approach in §4.1, and how it applies to our case studies involving failures and variable demands in §4.2 and §4.3 respectively. Although our framework is general, analysis of problem structure can substantially improve the quality of bounds, as we will show for the failure case study in §4.2. Finally, in §4.4, we compare our techniques with benchmarks drawn from the network management and robust optimization literature, and show that our techniques can obtain tighter bounds than these approaches.

4.1 Relaxing validation problems

Our approach works by relaxing the validation problems into more tractable LPs, and obtaining an upper bound on the worst-case link utilizations across scenarios. An optimization problem L is a relaxation of a problem N if every feasible solution in N can be mapped to a feasible solution in L, and the mapped solution's objective value in N is no better than that of its mapping in L.

Reformulation-Linearization Technique (RLT) [44] is a general approach to relax non-linear integer problems. The technique reformulates the problem by (i) adding new constraints obtained by taking products of existing constraints; and (ii) linearizing the resulting formulation by replacing monomials with new variables. For our problem (W), RLT can be constructed as long as $\alpha_{ij}(x)$, $\beta_{ik}(x)$ and $\gamma_k(x)$ are polynomial functions.

For example, consider a non-linear optimization problem where the objective is to minimize $xy-x+y$ subject to the constraints: (i) $(x-2)\geq 0$; (ii) $(3-x)\geq 0$; (iii) $(y-3)\geq 0$; and (iv) $(4-y)\geq 0$. Products of pairs of constraints are taken – e.g., the product of constraints (i) and (iii) results in a new derived constraint $(x-2)(y-3)\geq 0$, i.e., $xy-3x-2y+6\geq 0$. The product term xy is replaced by a new variable z. The objective is rewritten as $z-x+y$, and the derived constraint in the previous step expressed as $z-3x-2y+6\geq 0$. The resulting problem is linear, as it no longer has product terms. However, it is a relaxation in the sense that constraints (e.g., $z=xy$) that must be present to accurately capture the original problem are not included in the new problem.

The above represents the first step in a hierarchy of relaxations and the next steps involve multiplying more than two constraints and linearizing as discussed above. Further, the RLT hierarchy can be tightened using convex relaxations of monomials, which yield other well known hierarchies. As long as the set of inequalities in the verification problem define a bounded set, higher levels of this hierarchy of relaxations converge to the optimal value of the non-linear or integer program [27, 44]. Since the generated LPs can be large (more variables and constraints), we restrict attention to the first level of this hierarchy. Further, in practice, it often suffices to consider a subset of products even for the first level, which keeps the complexity of the resulting program manageable.

4.2 Validation across failure scenarios

Here, we discuss the RLT relaxation technique for our failure case study (formulation (F)). For concreteness, we consider all failure scenarios involving the simultaneous failure of f or fewer links. This failure model is used commonly in practice [50]. We discuss how to generalize the failure model later (§5). Incorporating this model results in replacing the constraint $x^f_{ij}\in X$ in (F) with the constraints $\sum_{\langle i,j\rangle\in E} x^f_{ij}\leq f$, and $x^f_{ij}\in\{0,1\}$.

Empirically, a simple RLT relaxation of the formulation does not yield a sufficiently tight upper bound to the validation problem. Instead, we reformulate the validation problem (F), and consequently derive constraints for the RLT relaxation, as described below:

Reformulating the validation problem. We add variables to (2), in a way that gives more flexibility in choosing solutions, but does not change the optimum. Adding variables to a primal results in additional constraints to the dual. Consequently, we derive constraints for (F) and the associated RLT relaxation LP, which are derived from the LP dual of (2), thus improving the bound on utilization. Specifically, we reformulate (2) as follows:

$$\begin{aligned}
&Uc_{ij}(1-x^f_{ij})+a_{ij}\geq \sum_t r_{ijt} \quad \langle i,j\rangle\in E\\
&\sum_j r_{ijt}-\sum_j r_{jit}=\begin{cases} d'_{it} & \forall t,i\neq t\\ -\sum_j d'_{jt} & \forall t,i=t\end{cases}\\
&r_{ijt},a_{ij}\geq 0 \quad \forall i,j,t\\
&d'_{it}=\begin{cases} d_{ij}+a_{ij} & \langle i,j\rangle\in E\\ d_{ij} & \langle i,t\rangle\notin E\end{cases}
\end{aligned} \tag{4}$$

We augment each link $\langle i,j\rangle$'s capacity with the extra

(variable) slack capacity a_{ij} for which we reserve the capacity along alternate paths in the network. In particular, the first constraint allows up to a_{ij} of the traffic on link $\langle i,j\rangle$ to be bypassed on the associated virtual link without counting it against the utilization of link $\langle i,j\rangle$. To compensate for this, we increase the total traffic that must be routed from i to j by a_{ij}, as indicated by the last constraint. It can be shown that (4) achieves the same optimal as (2). Further, because any feasible solution to (2) is also feasible to (4) (with slack variables a_{ij} being 0), (4) is more flexible in that it admits additional solutions.

Following the procedure outlined in Figures 1 and 2, this reformulated primal yields a reformulated validation problem (F') which consists of (F) with constraints $\lambda_{ij} \leq v_{ij} - v_{jj},\ \forall\langle i,j\rangle \in E$. Then, (F') simplifies to:

$$(G)\ \max \sum_{i,t} d_{it} v_{it}$$

$$v_{it} - v_{jt} \leq v_{ij} \quad \forall t, \langle i,j\rangle \in E \tag{5}$$

$$\sum_{\langle i,j\rangle \in E} v_{ij} c_{ij} (1 - x^f_{ij}) = 1$$

$$\sum_{\langle i,j\rangle \in E} x^f_{ij} = f \tag{6}$$

$$v_{it} \geq 0, v_{tt} = 0 \quad \forall i,t \tag{7}$$

$$x^f_{ij} \in \{0,1\}, \quad \langle i,j\rangle \in E \tag{8}$$

Proposition 1. *Reformulation (G) achieves the same optimal value as the original validation problem (F).*

The proof (see Appendix) shows that an optimal solution of (F') satisfies $v_{tt} = 0,\ \forall t$ and $v_{ij} = \lambda_{ij},\ \forall\langle i,j\rangle \in E$. The proposition then follows since (F) and (F') achieve the same optimal value having been derived respectively from primals (2) and (4) that achieve the same optimal.

Although (G) is non-linear because the product $v_{ij}x^f_{ij}$ is in the second constraint, we note that (G) has a finite objective only if the minimum cardinality edge-cut set of the topology contains more than f links, a condition that can be verified in polynomial time [18]. Moreover, we show (see Appendix) that if f failures cannot disconnect the nodes of the network, v_{ij} is bounded. Then, standard linearization of $v_{ij}x^f_{ij}$ that uses bounds on v_{ij} and $x^f_{ij} \in \{0,1\}$ reduces (G) to a mixed-integer linear program.

Relaxing the validation problem. Since the validation problem (G) is still intractable, we derive its first-level RLT relaxation as follows. First, the binary requirement $x^f_{ij} \in \{0,1\}$ is replaced by bound constraints, $x^f_{ij} \geq 0$ and $(1 - x^f_{ij}) \geq 0$. Next, the product of these bound constraints is taken with (5) and (7) and the product of (6) and (7) is taken. Finally, the nonlinear constraints are relaxed by introducing $vx^f_{iji'j'}$ to denote $v_{ij}x^f_{i'j'}$.

4.3 Validation across traffic demands

We now consider the tunnel selection case study (formulation (V)) and the problem of verifying utilization against uncertain demands. We discuss two models for specifying demands, and discuss the RLT relaxations.

Specifying demands. We consider two models:

• *Predicted demand:* This corresponds to scenarios when demands may be predicted from past history, a commonly used practice today. Consider optimizing the system for a set of known historical traffic matrices $\{d^h\}_{h\in H}$. As observed in [49], many predictors including the exponential moving average estimate the traffic matrix for a given interval as a convex combination of previously seen matrices. It may be desirable to verify the system for the convex hull of $\{d^1, d^2, \ldots d^h\}$, which ensures that all such predictors can be serviced with reasonable utilization. Specifically, this may be modeled by replacing the constraint $x^d \in X$ in (V) by the constraints $x^d = \sum_{h\in H} x_h d^h$, $x_h \geq 0$ and $\sum_{h\in H} x_h = 1$.

• *All demands that can be handled by the topology:* It may be desirable to understand the extent to which a topology must be over-provisioned if a tunneling solution is used compared to using an optimal MCF solution. This may be modeled by replacing the constraint $x^d \in X$ in (V) by the standard MCF constraints with x^d_{st} denoting demand from source s to destination t, and x^g_{ijt} a flow variable denoting traffic to t on link $\langle i,j\rangle$.

Obtaining the RLT relaxation. We obtain the RLT relaxation by taking the product of (i) inequalities involving v and λ variables with constraints of the form $x \geq 0$; (ii) inequalities involving x variables with constraints of the for $\lambda \geq 0$; (iii) inequalities involving v or λ with inequalities involving x; and (iv) equalities involving x variables with v variables.

4.4 Comparisons to alternate approaches

A key novelty of our framework is that it provides theoretical bounds on network performance across failures/demands, while allowing flexible adaptation. We can show that each RLT constraint we introduce in the problem makes the adaptations more flexible in a specific way. In contrast, prior theoretical work has focused on limited forms of adaptivity and we use them as benchmarks for our RLT relaxation approach. We show that our approach provides bounds that are at least as tight as these prior theoretical works, and later show empirically (§6) that the bounds are better in practice.

Oblivious approaches and generalizations. Oblivious routing [11, 28, 16, 9, 49] bounds utilizations across all links for a set of demands, while limiting how the network adapts to any given demand. While oblivious routing has mainly been considered in the context of MCF [9, 49], the oblivious approach applies to other networking contexts. For instance, in our tunneling case study, an *Oblivious Tunneling* formulation constrains y_{stk} (traffic on tunnel k from s to t) to be of the form $y_{stk} = \alpha_{stk} x^d_{st}$, where α_{stk} is invariant across demands.

The robust optimization literature has considered a more general form of adaptation than an oblivious approach, which can enable tighter bounds on worst-case link utilization [15]. Here, every variable y_i (e.g., each y_{stk} variable in our tunneling example) that a network determines for a given scenario x, is constrained to have the form $y_i = \alpha_{i0} + \sum_j \alpha_{ij} x_j$ where all α_{ij} coefficients must be invariant with x. Note that x_j variables capture scenario x (e.g., in our tunneling example, x is a traffic matrix, and each x_j is a cell in the matrix). In optimization terminology, y_i is an affine function of x. Note that an oblivious approach is a special case of affine policies where many of the α coefficients are zero.

We say the *linearity requirement* has been met when constraints $Y(x)$, and objective $F(x,y)$ are linear in (x,y). For example, in the tunneling case study, the constraints (3) and the objective, U, are linear in U, r, and x^d. Further, the conditions are satisfied by the original oblivious routing [9, 49], and while we do not elaborate, by other case studies such as routing with middleboxes. When network adaptation is restricted to affine policies, and the linearity requirement is met, an optimal set of α_{ij} coefficients may be computed efficiently using LP to minimize worst-case link utilizations [13]. We now state our result:

Proposition 2. *When the linearity requirement is met, an optimal affine policy can be efficiently computed. Under these circumstances, the first-level RLT relaxation for a validation problem is at least as tight as the bound from the optimal affine policy.*

The proof involves taking duals of the RLT relaxation. We do not elaborate on the technical details, and focus on the implications for validation. Further, for predicted demand (§4.3), Proposition 2 already implies that the first-level RLT can provide as tight a bound as an oblivious approach. However, we have shown a stronger result:

Proposition 3. *For the predicted demand case, the first-level RLT relaxation is an exact solution, while the oblivious solution may not always be exact.*

Some of our case studies do not satisfy the linearity requirement. In particular, the requirement is not satisfied for our case study involving failures (2) because the first constraint in (2) involves a non-linear term (product of U and x). Under these circumstances, an optimal affine policy may not be efficiently computable, and is thus not a viable benchmark. However, our framework is still applicable (as our failure case study has shown), since it only requires that the weaker condition that $Y(x)$ is linear in y variables for fixed x needs to be satisfied.

Benchmark for failure case study. R3 [50] tackles the validation problem under failures, but with the more limited goal of determining whether a network can handle all failures scenarios without congestion (i.e., whether $MLU \leq 1$), and with restrictions on how the network can adapt. R3 replaces failures with virtual demands (the traffic to be rerouted on failures) and computes an oblivious protection routing (MCF) for the virtual demand associated with each link. The formulation is only valid when $MLU \leq 1$, since the virtual demand on each link is assumed to not exceed the link capacity. In contrast, our formulation (G), and the associated first-level RLT relaxation is valid for any MLU, which can aid in tasks such as determining which failure scenarios are bad when the network is not sufficiently provisioned, and how best to augment link capacities to handle failures (§6.3). When $MLU \leq 1$, the bounds from R3 are conservative for our validation problem owing to the restriction on adaptations and since the impact of the failures is over-estimated. We have been able to show:

Proposition 4. *The first-level RLT relaxation of (G) provides at least as tight a bound as R3, whenever R3 provides a valid bound.*

In fact, we can impose similar restrictions as R3 on how traffic is rerouted in response to failures by appropriately choosing a subset of RLT constraints. Yet, the MLU will reduce because we optimally chose slack a_{ij} instead of assuming it is c_{ij}. The proof of Proposition 4 considers a special affine policy for $y = (r, U, a)$ in (4), where U does not adapt with x and $a_{ij} = \alpha_{ij} x_{ij}$. We show that all such policies that yield $U \leq 1$ can be made feasible to R3, and, therefore, the bound for R3 is no better than the one obtained with this policy restriction. Since RLT encompasses search over these policies, the result follows. We will show in §6 that RLT yields tighter bounds than R3 whenever the network utilization is less than 1.

5 Aiding synthesis and generalizations

§4 has shown how our framework applies to two validation case studies. We next discuss applications to robust design (§5.1), and to other validation problems (§5.2).

5.1 Augmenting capacities to bound utilization

To see how our validation framework can help in robust design, consider the problem of incrementally adding capacity to existing links to ensure all failure scenarios of interest can be handled (with $U \leq 1$), while minimizing the costs of augmented capacity. We can extend (1) to model the capacity augmentation problem as follows:

$$\min_{\delta \geq 0} \max_{x^f \in X} \min \left\{ \sum_{\langle i,j \rangle \in E} w_{ij} \delta_{ij} \;\middle|\; \begin{array}{l} (c_{ij} + \delta_{ij})(1 - x^f_{ij}) \geq \sum_t r_{ijt} \\ r \text{ is a routing for } d \end{array} \right\}$$

where X is the set of failure scenarios, and δ_{ij} and w_{ij} are respectively the incremental capacity added to link $\langle i,j \rangle$, and the cost per unit capacity. Further, r is a routing for d if r satisfies the flow balance constraints of an MCF formulation. Then, dualizing the inner minimization problem results in a two-stage formulation whose

inner problem is an IP since X is a discrete set. However, using the RLT relaxation technique presented in our framework, we replace the inner problem by an upper-bounding LP which can be dualized to upper-bound the cost of augmentation. This yields an LP based approach to conservatively augmenting capacity.

The above discussion also motivates an iterative approach to design. At each iteration, we solve a capacity augmentation problem considering failure scenarios identified in earlier rounds. Then, with the new capacities, we solve the failure validation problem to identify additional failure scenarios and iterate. At any stage, this provides a lower bound on the optimal capacity augmentation. Although the iterative procedure works well empirically for capacity augmentation, in other robust design problems, finding the worst uncertainty may be hard and the procedure may require too many iterations. In contrast, the LP based approach presented above always yields a conservative robust design quickly.

5.2 More general validation problems

In this section, we discuss how our framework can tackle other validation problems beyond our case studies.

Routing with middlebox constraints. Our framework may be used to obtain bounds on MLU when routing is constrained to satisfy middlebox policies [39, 47, 7]. The requirement that traffic from s to t be routed across a series of middleboxes can be modeled by associating each flow with a state variable which indicates a given middlebox has been traversed. The state is modified by each middlebox on the path. (2) is reformulated by introducing variables $r_{ijst\phi}$ which denote the flow on link $\langle i,j\rangle$ from s to t and for packets with state ϕ, and appropriately modifying the flow balance equations, and capacity constraints. The validation problem may now be formulated and solved across failures, or demands using the same approach as our two case studies.

Simultaneously varying failures and demands. We may desire to ensure utilizations are acceptable across any combination of failures and demands. This can be achieved by directly taking (F), and replacing demand variables d_{st} with variables x^d_{st}, and adding constraints for both x^d and x^f using previously studied models. A similar RLT relaxation applies in this case as well.

Handling shared risk link groups (SRLGs). We have considered a model where at most f links fail simultaneously. In practice, multiple links may fail together (e.g., a fiber cut may impact all links in the affected fiber bundle) [48]. The set of link groups G is considered, and each group g is associated with a set of links that may fail together. We introduce variables x^f_g which indicates whether a particular link group has failed. The validation problem is modeled by considering formulation (F), and replacing the constraints $x^f \in X$ with the constraints $x^f_{ij} = 1 - \prod_{\langle i,j\rangle \in g, g\in G}(1-x^f_g)$, where all x^f_{ij} and x^f_g variables are binary, and $\sum_{g\in G} x^f_g \leq f$. This captures that link $\langle i,j\rangle$ has failed iff any group that it belongs to has failed, and at most f link groups may fail simultaneously. To eliminate the product terms, the first constraint can be linearized with the constraints $x^f_{ij} \geq x^f_g, \langle i,j\rangle \in g, g \in G$, and the constraint $x^f_{ij} \leq \sum_{\langle i,j\rangle \in g, g\in G} x^f_g$. An RLT relaxation may now be applied as normal. Alternately, other linearized constraints can be derived from exploiting this relationship within the RLT scheme that we do not detail.

Network	Nodes	Edges	Date	Link Capacity
Abilene	11	28	2004	homogeneous
ANS	18	50	2011	homogeneous
GEANT	41	118	2014	heterogeneous

Table 1: Topologies

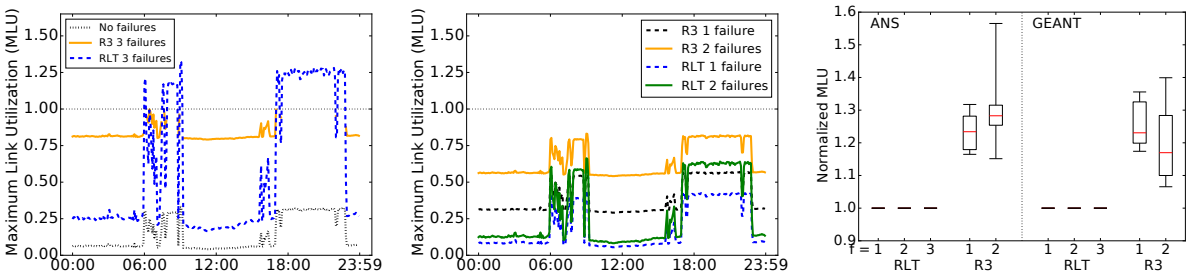
6 Evaluation

We evaluate the effectiveness of our framework in validating topology design under failures (§6.1), and tunnel selection under variable demands (§6.4). We compare our performance bounds with those obtained using existing approaches. Further, we show we can (i) identify bad failure scenarios (§6.2), (ii) optimally augment network capacity to handle failures (§6.3), and (iii) evaluate common design heuristics for tunnel selection (§6.4).

We evaluate our work using real topologies obtained from the Internet TopologyZoo [6]. We focus on three topologies: Abilene, ANS and GEANT [2] (Table 1), where Abilene and ANS have homogeneous link capacities, and GEANT has heterogeneous link capacities. All our LPs and IPs were run using CPLEX [3] (version 12.5.1.0). Our primary performance metric is MLU (§2.2) though we also consider how MLU impacts latency through emulation on an SDN testbed (§6.2).

6.1 Validation across failure scenarios

We evaluate the efficacy of our approach for determining MLU across failure scenarios, comparing MLU bounds produced by our RLT-based LP (§4.2) with (i) the IP (G) which can determine the optimal MLU value (§4.2); and (ii) R3 [50] (§4.4), the best known current approach. (G) is an intractable problem used only for comparison, and the running time of both our RLT relaxation and (G) is shown at the end of this section. We report the MLU returned by the R3 formulation instead of just the binary decision of whether $MLU \leq 1$ used in the original work. Recall R3 only provides valid bounds on MLU when $MLU \leq 1$ (§4.4). We study failure scenarios involving f arbitrary link failures, f ranging from 1 to 3, which practitioners indicated were important to consider. To ensure connectivity after multiple failures, we eliminated one-degree nodes from ANS and GEANT topologies, and modeled each edge as consisting of 2 sub-links



(a) Abilene: baseline and 3 failures (b) Abilene: 1 and 2 failures (c) ANS and GEANT: normalized MLU

Figure 4: Validation across failure scenarios, comparing RLT and R3, for various topologies.

# of Failures	RLT (sec)	IP (sec)	% IP Completed
1	640.58	60.50	100
2	622.60	394.97	100
3	607.68	3890.16	60
4	598.31	–	0
5	586.79	–	0

Table 2: Average running time of the RLT scheme and the optimal IP for GEANT. For the optimal IP, the average running time is computed with instances that completed in 2 hours.

of equal capacity for all topologies. The resulting ANS (GEANT) network has 17 (32) nodes and 96 (200) edges.

We begin by presenting results with the Abilene topology using real traffic data [1]. Figure 4a shows the MLU for $f = 3$, for the RLT and R3 schemes for all traffic matrices measured on April 15th, 2004, a day which experienced a wide variety of traffic patterns. The MLU under normal conditions (no failures) is shown as a baseline. The RLT scheme matches the optimal IP scheme for all traffic matrices, and hence we do not present the IP scheme. The graph shows that several traffic matrices stress the network to achieve $MLU > 1$, indicating it is not provisioned to handle all three simultaneous link failures. Further, the RLT scheme achieves a tighter bound than R3 for all cases where $MLU \leq 1$, and unlike R3, it can provide valid bounds even when $MLU \geq 1$.

Figure 4b presents results for Abilene, but for $f = 1$ and 2. Again, the optimal IP is not shown, since RLT matches optimal. The graph shows the MLU is under 1 for all matrices, indicating the network can handle all possible 2 link failures. Moreover, RLT achieves a tighter bound on MLU than R3 for all matrices. We repeat the experiments with ANS and GEANT topologies. Since actual traffic matrices were not available to us, we generated multiple traffic matrices for each topology using the gravity model [52]. The traffic matrices were chosen so as to keep the link utilizations between 0.3 and 0.45 under normal conditions. Figures 4c presents the normalized MLU for R3 and RLT, relative to the optimal IP for each f. Boxplots depict variation across the matrices. The graph shows that for all f and all traffic matrices, RLT always achieves a normalized MLU of 1, indicating it always matches optimal. The normalized MLU with R3 is higher, e.g., ranging from 1.15 to 1.57 for ANS $f = 2$. Note that results for R3 are not shown for $f = 3$ because all traffic matrices with GEANT, and all but 2 matrices with ANS achieved an optimal MLU above 1, indicating the network was not sufficiently provisioned for them. In contrast, the optimal MLU was under 1 for $f = 1$ and 2, for both topologies, and all traffic matrices.

A surprising aspect of our results is that across all topologies and traffic matrices, the RLT scheme matches the optimal IP. We have also investigated this further for other synthetic topologies and other settings, and have found RLT to match optimal across all the examples. We leave to future work further investigation of whether the first-level RLT in fact can be proven to match the optimal for this case study, or if counter-examples exist.

Running time. We report the running time (Table 2) from experiments with GEANT, the largest topology in our set, on a machine with 8-core 3.00 GHz Intel Xeon CPU and 94 GB memory. To create an even larger topology, we modeled each edge as consisting of 10 sub-links of equal capacity. The resulting network has 32 nodes and 1000 edges. Table 2 shows the average running time of RLT and the optimal IP using 10 traffic matrices generated by the gravity model. Since many IP instances didn't finish even after several hours, we set a 2-hour limit to the solver. Results show that the running time stays stable for RLT, but explodes for the optimal IP, as the number of failures increases. At $f = 3$, 40% of the IP instances did not converge. At $f = 4$ and 5, none of the IP instances converged, and the gaps[1] are larger than 0.5 in all the cases, indicating that the IP solutions found by the solver within 2 hours are still far from the optimal.

6.2 Impact of failures on application performance

Our validation framework can be used to identify failure scenarios that result in high MLU, which could then be emulated on a network testbed to study application performance metrics such as latency under such scenarios.

[1] gap = (UB - LB) / UB, where UB and LB denote the upper bound and the lower bound of the optimal objective value.

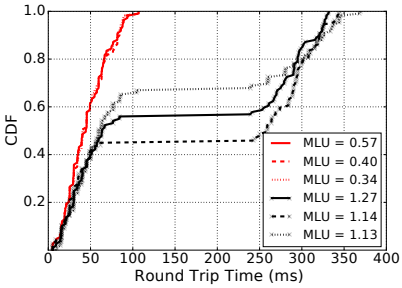


Figure 5: Latency CDF of different failure scenarios.

Finding bad failure scenarios. In general, it is hard to find failure scenarios for the original validation problem (G) that result in a high MLU since it is an IP. A random search is inefficient – e.g., for a certain Abilene traffic matrix, a brute-force search revealed only 0.05% of 3-failure scenarios achieved $MLU > 1$, while 0.08% cases achieved $MLU > 0.8$.

We use a branch and bound algorithm leveraging our RLT LP relaxation. At each exploration step, the failure status of a subset of links is fixed at each node (in the initial step, none of the links are fixed), and the relaxation LP is run to determine a (possibly fractional) solution that results in the highest MLU for the LP. The link with the highest fractional failure (say $\langle i, j \rangle$) is considered, and the LP is rerun fixing x_{ij}^f as each of 0 and 1. Branches where the $MLU < 1$ are pruned. Of the remaining candidate unexplored nodes, the node with the highest MLU is visited. Ties are broken by picking the node at the lowest level in the search tree. The process is run until an integral solution is found, and the search procedure could be continued to determine multiple integral solutions. If the LP relaxation is tight, the search procedure solves at most as many LPs as the number of edges in the topology to find a failure scenario that results in the highest MLU, and our empirical experiments show it takes much fewer steps in practice.

Emulation on an SDN testbed. We emulated the Abilene topology on Mininet [4]. Traffic was generated using the Ostinato traffic generator [5], and an actual Abilene traffic matrix snapshot. We used the procedure above together with our validation framework to identify multiple failure scenarios where MLU exceeded 1. Figure 5 presents measured Round Trip Time (RTT). Each curve corresponds to a failure scenario, and shows a CDF of the median RTT across all source-destination pairs for that scenario. The three curves to the right (black) represent failure scenarios identified by our framework with $MLU > 1$. To contrast, we show three other randomly generated 3-link failure scenarios with lower MLU (red, and to the left – note the curves overlap). The results illustrate that RTTs are significantly higher for the high MLU scenarios identified by our framework.

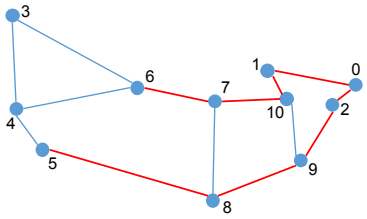


Figure 6: Abilene with links augmented shaded in red.

6.3 Deriving valid capacity augmentations

Our robust validation framework also guides operators in how best to augment link capacities to guarantee $MLU < 1$ across failure scenarios. As discussed in §5, our framework can be applied in an iterative approach that achieves optimal, or may be formulated as a single LP that does not guarantee optimality but solves efficiently.

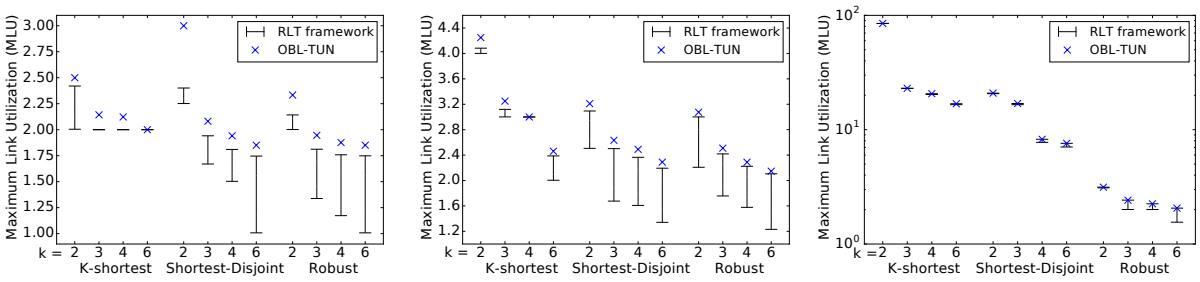
Table 3 illustrates the iterative procedure for an Abilene traffic matrix under three simultaneous link failures. Recall that each iteration consists of (i) a validation step, which either certifies $MLU \leq 1$ for the topology (augmented by capacity increase suggested in prior iteration), or identifies a violating failure scenario; and (ii) an augmentation step, which identifies minimum capacity augmentation needed to handle all failure scenarios identified in prior iterations. The procedure terminates when the validation step certifies $MLU \leq 1$. The augmentation step is a small variant of (2) (see Appendix for details). It is an LP and can easily incorporate practical constraints that limit which links can have their capacity augmented.

We have also formulated the problem as an LP with the stricter requirement that the RLT relaxation of the validation problem achieves $MLU \leq 1$ (§5). The LP achieves the same optimal augmentation as the iterative approach above, which is not surprising given that in all instances we have tried the integrality gap has been 1. More generally, the design LP yields an augmentation cost no worse than $\alpha\text{OPT} + (\alpha - 1)\text{BASE}$, where OPT is the optimal augmentation cost, BASE is the cost of the base network and α is the integrality gap of the RLT relaxation.

6.4 Validation across traffic demands

We next consider how our approach can validate that utilizations are acceptable across demands, focusing on the tunneling case study. For each topology, we consider tunnels pre-selected using the following strategies:

Non-robust strategies. These strategies pick tunnels without explicitly considering tolerance to a range of demands. Specifically, we consider: (i) *K-shortest:* Here, the K shortest paths between each source and destination pair are chosen. Prior works [30, 29] have used this approach to generate an initial candidate set of tunnels, and [30] ultimately picks a subset in a demand-sensitive manner; (ii) *Shortest-Disjoint:* Here, the shortest path is selected. Among other paths, one that overlaps the least with prior choices is selected in an iterative fash-



(a) Abilene (b) ANS (c) GEANT

Figure 7: MLU of RLT framework and Oblivious Tunneling (OBL-TUN) for different tunnel designs and topologies.

ion. Combining path lengths and disjointness is a natural approach to tunnel selection [46].

Robust strategies. We also consider a heuristic called *Robust*, which derives tunnels by decomposing the optimal oblivious routing [25] (details in the Appendix). Since oblivious routing derives an MCF that performs well across all demands, tunnels derived from such a flow have the potential to perform well across demands.

For each tunnel selection approach, our goal is to determine MLU with an adaptive strategy, where traffic is split optimally across tunnels for each demand by solving (3). Since the associated validation problem is nonlinear, we obtain bounds on MLU using (i) our RLT-based framework and (ii) an *Oblivious Tunneling* formulation (abbreviated as OBL-TUN) which minimizes MLU across all demands under the constraint that the fraction of traffic on each tunnel cannot vary with demand (§4.4). While both the RLT framework and OBL-TUN provide upper-bounds on the actual MLU, our framework can also be used to derive a lower bound. Specifically, we solve (V) after fixing the demand to the worst-performing demand for the RLT (or oblivious) relaxation. While this already provides a lower bound, we improve the initial lower bound using a local search procedure on (V), which involves alternating minimization on (v, λ) and x^d. These are tractable problems since (V) is linear if either (v, λ) or x^d are fixed.

Results. We evaluate a total of six schemes, combining our three tunnel selection heuristics with the two ways to obtaining bounds on MLU. For the set of demands, we consider all demands that can be routed with given capacities (§4.3). An MLU higher than 1 indicates the amount of over-provisioning required if tunneling were used to support all demands that the topology could handle with MCF routing.

Figures 7a, 7b and 7c present the MLU across all traffic demands for each of three strategies and three topologies, and different number of selected tunnels (K). Each cross shows the upper bound determined by OBL-TUN, while the vertical bar shows the upper and lower bounds obtained with our RLT-based framework. For GEANT, our current RLT implementation had a high memory requirement that can be addressed using standard decomposition techniques [36] in the future – hence we only report upper bounds achieved by OBL-TUN.

Step	Counter Examples	MLU	Total New Capacity
1	$(1,10,H)$, $(2,9,F)$	1.274	2.744 Gbps
2	$(2,9,H)$, $(1,10,F)$	1.274	5.488 Gbps
3	$(9,8,H)$, $(10,7,F)$	1.217	7.653 Gbps
4	$(10,7,H)$, $(9,8,F)$	1.217	9.818 Gbps
5	$(0,2,H)$, $(1,10,F)$	1.192	11.743 Gbps
6	$(1,0,H)$, $(1,10,F)$	1.071	12.452 Gbps
7	$(7,6,H)$, $(8,5,F)$	1.006	12.509 Gbps
8	$(8,5,H)$, $(7,6,F)$	1.006	12.566 Gbps
9	–	1.000	–

Table 3: Iterative optimal capacity augmentation for Abilene (Figure 6). Each row shows MLU and counter example generated by the validation step, and the total capacity that must be added across all links as per the augmentation step to address all prior counter-examples. H (F) indicates one (both) sub-link(s), (each initially 5 Gbps) associated with the edge fails.

Several points can be made. First, our RLT framework often obtains tighter upper bounds than OBL-TUN strengthening Proposition 2. For example, for Abilene with $K = 2$, and Shortest-Disjoint tunnel selection, the upper bounds with OBL-TUN and the RLT framework are 3 and 2.4 respectively. Second, by providing lower bounds as well, our framework can exactly solve the non-linear problem (V) in quite a few cases. For instance for Abilene and the K-shortest heuristic, a single horizontal line is shown for $K = 3$ and higher, indicating that our framework can determine the optimal MLU.

Third, through a combination of lower and upper bounds, our framework can provide valuable insights on tunnel selection heuristics used by system practitioners. For example, for K-shortest $K = 6$, not only does our framework determine exact MLU, but also the MLU is the same as OBL-TUN. This indicates that when tunnels are selected using K-shortest, adapting how traffic is split across tunnels with demand performs no better than a non-adaptive approach. The trend is particularly pronounced for GEANT where our framework indicates the lower-bounds on MLU are higher than 16 even for $K = 6$, and very close to the oblivious solution. While

recent work has suggested picking the K shortest tunnels and then picking a subset in a demand-sensitive manner [30], this result shows the possibility for this heuristic to perform poorly under certain demand patterns. The Shortest-Disjoint heuristic performs much better for Abilene and ANS, but performs poorly for GEANT – for $K = 6$, the lower bound is 7.05, close to the MLU of 7.59 with an oblivious approach.

While the non-robust design strategies perform poorly, *Robust* performs much better. The benefits are particularly stark for GEANT, e.g., for $K = 6$, the MLU ranges between 1.54 and 2.05. We have also experimented with robust tunnels and predicted demands obtained from real traffic matrices that are scaled so as to stress the Abilene network. The RLT framework achieves the optimal MLU (Proposition 3). OBL-TUN however results in MLU that is 6.84 times worse than optimal. Overall, these results show the value of our RLT-based framework.

7 Related work

Like work on network verification (e.g., [34, 33]), robust validation ensures that network designs meet operator intent. While verification efforts have focused on correctness of the network data-plane, and switch configurations, robust validation is an early attempt at verifying quantifiable network properties. Our framework complements topology synthesis tools [43] by allowing specification of robust design requirements, and providing the underlying optimization substrate.

Prior work on traffic engineering has focused on adaptive settings [20, 32] or has derived a robust routing that optimizes for multiple demands assuming that the routing does not change across demands [9, 8, 49, 51]. Robust routing schemes include oblivious schemes which do not use prior traffic data (e.g., [9, 8]), that route based on multiple historical traffic matrices (e.g., [51]), and those that combine these techniques [49]. Oblivious schemes arose from pioneering work in the theoretical computer science community [40, 41]. In contrast, we obtain worst-case utilization bounds for network designs, where topology and tunnels are invariant, but routing may adapt in practical yet richer ways. It has been shown that adaptive tunnels may, in the worst-case, not benefit much relative to oblivious routing [25]. Instead, we show that provable gains are achieved for specific topologies which have also been observed in practice [35].

Several works have looked at traffic engineering in the presence of failures [8, 48, 50, 37], and we have extensively compared our work with [50]. [8, 48] studied partial adaptation to failures as a way to balance flexible adaptation with the cost for adaptation. [37] optimizes bandwidth assignments to flows, guaranteeing that no congestion occurs with failures. While we do not elaborate, this model can be expressed using our framework. Prior work [22] developed ways to choose OSPF weights which are robust to single link failures. In contrast, we allow flexible adaptation, minimize MLU, and aid robust design of networks that cope well with failures.

Many recent works have looked at how traffic must be routed in the presence of middleboxes (e.g., [39, 47, 7]). There is a growing trend for virtualization of middleboxes, which may allow placements to change on the fly [45, 7]. Our framework can accommodate problems that adapt routing to handle uncertain demands/failures while satisfying middlebox constraints both for fixed placements, and when allowing placements to adapt along with routing.

Beyond networking, the complexity status of robust optimization formulations has been investigated and tractable formulations derived for various special cases [12, 14]. Recent literature has considered limited adaptability in robust binary programming applications including supply chain design and emergency route planning [26, 10]. Instead, our work considers more general forms of adaptivity, focuses on the networking domain, and brings relaxation hierarchies from non-convex optimization to bear on robust optimization problems.

8 Conclusions

In this paper, we have made three contributions. First, we have presented a general framework that network architects can use to validate that their designs perform acceptably across a (possibly exponential and non-enumerable) set of failure and traffic scenarios. Second, by explicitly modeling richer ways in which networks may adapt to failures, and traffic patterns, we have obtained tighter bounds on MLU than current theoretical tools, which consider more limited forms of adaptation for tractability reasons. Third, we have demonstrated the practical applicability of our framework. While the first-level RLT can provably solve the validation problem for predicted demand, surprisingly, it also determines optimal MLU for for all our experiments with the failure case study. Empirical results confirm that our techniques consistently out-perform oblivious methods that can be unduly conservative. Finally, our framework can enable operators to understand performance under failures, guide incremental design refinements, and shed new light on commonly accepted design heuristics. Our initial results encourage us to explore larger networks, study the quality of bounds on other validation problems, and consider network design more extensively in the future.

Acknowledgements

We thank our shepherd Nate Foster, and the reviewers for their insightful feedback. This work was supported in part by the National Science Foundation (Award Number 1162333), and by a Google Research Award.

A Appendix

Proof of Proposition 1: Clearly, the optimal value of (G) is no more than that of (F') because (G) has the following additional constraints (i) for all $\langle i,j\rangle \in E$, $\lambda_{ij} = v_{ij}$, and (ii) and for all nodes t, $v_{tt} = 0$. Therefore, we only need to show that the optimal value of (F') is no more than that of (G). Let (λ^*, v^*, x^{f*}) be optimal in (F'). Denote by $\mathrm{SP}_{it}(\lambda)$ the shortest path between i and t with edge-lengths λ. For any path P_{it} connecting nodes i and t, it follows from the first constraint in (F) that $v^*_{it} - v^*_{tt} \leq \sum_{\langle i,t\rangle P_{it}} \lambda^*_{ij}$ and, so, minimizing rhs over paths yields $v^*_{it} - v^*_{tt} \leq \mathrm{SP}_{it}(\lambda^*)$. For any link $\langle i,j\rangle$ this implies that $\lambda^*_{ij} \leq v^*_{ij} - v^*_{jj} \leq \mathrm{SP}_{ij}(\lambda^*) \leq \lambda^*_{ij}$, where the first inequality is from the slack-induced constraint, the second inequality follows from discussion above, and the third inequality because $\langle i,j\rangle$ is a valid path from i to j. Therefore, equality holds throughout. Now, consider the solution (v', x^{f*}) such that $v'_{it} = \mathrm{SP}_{it}(\lambda^*)$. We show that this solution is feasible to (G). Clearly, $v'_{it} - v'_{jt} \leq v'_{ij}$ because the shortest path from j to t can be augmented with $\langle i,j\rangle$ to yield a path from i to t. Next, because $v'_{ij} = \mathrm{SP}_{ij}(\lambda^*) = \lambda^*_{ij}$, where the last equality was shown above, it follows that $\sum_{\langle i,j\rangle} v'_{ij} c_{ij}(1 - x^{f*}_{ij}) = 1$. Moreover, $v'_{it} = \mathrm{SP}_{it}(\lambda^*) \geq 0$ because $\lambda^*_{ij} \geq 0$ and, trivially, $v'_{tt} = \mathrm{SP}_{tt}(\lambda^*) = 0$. Therefore, (v', x^{f*}) is feasible to (G). Finally, $\sum_{i,t} d_{it} v'_{it} = \sum_{i,t} d_{it} \mathrm{SP}_{it}(\lambda^*) \geq \sum_{i,t} d_{it}(v^*_{it} - v^*_{tt})$, where the equality follows from the definition of v' and the inequality by summing products of $\mathrm{SP}_{it}(\lambda^*) \geq (v^*_{it} - v^*_{tt})$ with $d_{it} \geq 0$. Therefore, the optimal value of (G) is at least as large as that of (F').

Proof that (G) can be formulated as an Integer Program after a polynomial time verification of graph connectivity: The objective of (G) is not finite if the minimum edge-cut set contains f or fewer links, a fact that can be verified in polynomial time [18]. Now consider that the topology is not disconnected after any simultaneous set of f link failures. We show that $v_{it} \leq \frac{1}{c_{min}}$, where $c_{min} = \min_{\langle i,j\rangle \in E} c_{ij}$. To prove the bounds, let NF denote the set of links that do not fail when the optimal value of (G) is achieved. For any pair of nodes i and t, there exists a path (whose edges we denote as P) on the failure of this set of links. By adding the first constraint of (G) for all edges along P, $v_{it} = \sum_{\langle i,j\rangle \in P} v_{ij} \leq \sum_{\langle i,j\rangle \in NF} v_{ij}$. From the second constraint of (G), $\sum_{\langle i,j\rangle \in NF} v_{ij} c_{ij} = 1$, and hence $\sum_{\langle i,j\rangle \in NF} v_{ij} \leq 1/c_{min}$. The bounds follow.

Multiplying the bound constraints $0 \leq v_{ij} \leq \frac{1}{c_{\min}}$ with x^f_{ij} and $1 - x^f_{ij}$ allows us to linearize the above mixed-integer non-linear program into an integer program. This is achieved by replacing $v_{ij} x^f_{ij}$ with a new variable $v x^f_{ij}$ and observing that it is automatically constrained to be $v_{ij} x^f_{ij}$ when $x^f_{ij} \in \{0,1\}$.

Capacity augmentation procedure. For a given scenario, the capacity augmentation problem is easy to model and solve as a linear program. Specifically, (2) is modified by setting the utilization bound $U = 1$, and replacing capacity c_{ij} with $c_{ij} + \delta_{ij}$, where δ_{ij} is the incremental capacity that must be added to link $\langle i,j\rangle$. The objective is $\sum_{ij} w_{ij}\delta_{ij}$, where w_{ij} is the cost associated with each unit of capacity added to link $\langle i,j\rangle$. The formulation is easily extended to multiple scenarios, by replicating the set of constraints (2) modified as above, for each scenario. Practical cabling constraints that constrain which links can have their capacity augmented and by how much are easily incorporated by adding bounds to δ_{ij}.

Robust tunnel design heuristic. To generate a set of tunnels by decomposing the optimal oblivious routing, a derived graph is considered which has the same nodes and edges as the original topology, but with each edge having a weight equal to the flow from the oblivious routing. The widest path (the path with the highest bottleneck link capacity) is chosen as a tunnel. The bottleneck capacity of this path is now decremented from all other edges on this path in the derived graph. This procedure is repeated until k tunnels are obtained.

References

[1] Abilene traffic matrices. `http://www.cs.utexas.edu/~yzhang/research/AbileneTM/`.

[2] GEANT network. `http://geant3.archive.geant.net/Network/NetworkTopology/pages/home.aspx`.

[3] IBM ILOG CPLEX optimization studio. `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer`.

[4] Mininet. `http://mininet.org/`.

[5] Ostinato network traffic generator and analyzer. `http://ostinato.org/`.

[6] Topology zoo. `http://www.topology-zoo.org/`.

[7] B. Anwer, T. Benson, N. Feamster, and D. Levin. Programming slick network functions. In *Proceedings of ACM SIGCOMM Symposium on Software Defined Networking Research*, pages 14:1–14:13, 2015.

[8] D. Applegate, L. Breslau, and E. Cohen. Coping with network failures: Routing strategies for optimal demand oblivious restoration. In *Proceedings of ACM Special Interest Group on Measurement and Evaluation (SIGMETRICS)*, pages 270–281, 2004.

[9] D. Applegate and E. Cohen. Making intra-domain routing robust to changing and uncertain traffic demands: Understanding fundamental tradeoffs. In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 313–324, 2003.

[10] P. Awasthi, V. Goyal, and B. Y. Lu. On the adaptivity gap in two-stage robust linear optimization under uncertain constraints. `http://www.columbia.edu/~vg2277/column-wise.pdf`, 2015.

[11] Y. Azar, E. Cohen, A. Fiat, H. Kaplan, and H. Räcke. Optimal oblivious routing in polynomial time. *J. Comput. Syst. Sci.*, 69(3):383–394, 2004.

[12] A. Ben-Tal, L. E. Ghaoui, and A. Nemirovski. *Robust Optimization*. Princeton University Press, Princeton, NJ, 2009.

[13] A. Ben-Tal, A. Goryashko, E. Guslitzer, and A. Nemirovski. Adjustable robust solutions of uncertain linear programs. *Mathematical Programming*, 99(2):351–376, 2004.

[14] D. Bertsimas, D. B. Brown, and C. Caramanis. Theory and applications of robust optimization. *SIAM Review*, 53(3):464–501, 2011.

[15] D. Bertsimas and V. Goyal. On the power and limitations of affine policies in two-stage adaptive optimization. *Mathematical programming*, 134(2):491–531, 2012.

[16] M. Bienkowski, M. Korzeniowski, and H. Räcke. A practical algorithm for constructing oblivious routing schemes. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 24–33, 2003.

[17] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A retrospective on evolving sdn. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software-Defined Networks*, pages 85–90, 2012.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press Cambridge, 2001.

[19] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ, 1963.

[20] A. Elwalid, C. Jin, S. Low, and I. Widjaja. MATE: MPLS adaptive traffic engineering. In *Proceedings of IEEE INFOCOM*, pages 1300–1309, 2001.

[21] S. Even, A. Itai, and A. Shamir. On the complexity of time table and multi-commodity flow problems. In *Proceedings of Symposium on Foundations of Computer Science*, pages 184–193, 1975.

[22] B. Fortz, M. Thorup, et al. Robust optimization of OSPF/IS-IS weights. In *Proceedings of International Network Optimization Conference*, pages 225–230, 2003.

[23] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 58–72, 2016.

[24] A. Gupta, M. T. Hajiaghayi, and H. Räcke. Oblivious network design. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 970–979, 2006.

[25] M. T. Hajiaghayi, R. D. Kleinberg, and T. Leighton. Semi-oblivious routing: Lower bounds. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, 2007.

[26] G. A. Hanasusanto, D. Kuhn, and W. Wiesemann. K-adaptability in two-stage robust binary programming. *Operations Research*, 63(4):877–891, 2015.

[27] D. Handelman. Representing polynomials by positive linear functions on compact convex polyhedra. *Pacific Journal of Mathematics*, 132(1):35–62, 1988.

[28] C. Harrelson, K. Hildrum, and S. Rao. A polynomial-time tree decomposition to minimize congestion. In *Proceedings of Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 34–43, 2003.

[29] V. Heorhiadi, M. K. Reiter, and V. Sekar. Simplifying software-defined network optimization using SOL. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*, pages 223–237, 2016.

[30] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 15–26, 2013.

[31] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4:

Experience with a globally-deployed software defined wan. In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, 2013.

[32] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 253–264, 2005.

[33] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*, pages 113–126, 2012.

[34] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*, pages 15–27, 2013.

[35] P. Kumar, Y. Yuan, C. Yu, N. Foster, R. Kleinberg, and R. Soulé. Kulfi: Robust traffic engineering using semi-oblivious routing. *arXiv:1603.01203 [cs.NI]*, 2016.

[36] C. Lemaréchal, A. Nemirovskii, and Y. Nesterov. New variants of bundle methods. *Mathematical programming*, 69(1-3):111–147, 1995.

[37] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter. Traffic engineering with forward fault correction. In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 527–538, 2014.

[38] R. Potharaju and N. Jain. When the network crumbles: An empirical study of cloud network failures and their impact on services. In *Proceedings of ACM Annual Symposium on Cloud Computing*, pages 15:1–15:17, 2013.

[39] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simple-fying middlebox policy enforcement using sdn. In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 27–38, 2013.

[40] H. Räcke. Minimizing congestion in general networks. In *IEEE Symposium on Foundations of Computer Science*, pages 43–52, 2002.

[41] H. Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *Proceedings of Annual ACM Symposium on Theory of Computing*, pages 255–264, 2008.

[42] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture. *RFC 3031*, 2001. `https://tools.ietf.org/html/rfc3031`.

[43] B. Schlinker, R. N. Mysore, S. Smith, J. C. Mogul, A. Vahdat, M. Yu, E. Katz-Bassett, and M. Rubin. Condor: Better topologies through declarative design. In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 449–463, 2015.

[44] H. D. Sherali and W. P. Adams. *A reformulation-linearization technique for solving discrete and continuous nonconvex problems*. Springer Science & Business Media, Dordrecht, 1999.

[45] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 13–24, 2012.

[46] D. Sidhu, R. Nair, and S. Abdallah. Finding disjoint paths in networks. In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 43–51, 1991.

[47] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. In *Proceedings of ACM CoNEXT Conference*, pages 213–226, 2014.

[48] M. Suchara, D. Xu, R. Doverspike, D. Johnson, and J. Rexford. Network architecture for joint failure recovery and traffic engineering. In *Proceedings of ACM Special Interest Group on Measurement and Evaluation (SIGMETRICS)*, pages 97–108, 2011.

[49] H. Wang, H. Xie, L. Qiu, Y. R. Yang, Y. Zhang, and A. Greenberg. COPE: Traffic engineering in dynamic networks. In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 99–110, 2006.

[50] Y. Wang, H. Wang, A. Mahimkar, R. Alimi, Y. Zhang, L. Qiu, and Y. R. Yang. R3: Resilient routing reconfiguration. In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 291–302, 2010.

[51] C. Zhang, Z. Ge, J. Kurose, Y. Liu, and D. Towsley. Optimal routing with multiple traffic matrices tradeoff between average and worst case performance. In *Proceedings of International Conference on Network Protocols*, 2005.

[52] Y. Zhang, Z. Ge, A. Greenberg, and M. Roughan. Network anomography. In *Proceedings of ACM Internet Measurement Conference*, pages 30–30, 2005.

Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads

Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett,
Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao,
Anirudh Sivaraman, George Porter, Keith Winstein

Stanford University, University of California San Diego, Massachusetts Institute of Technology

Abstract

We describe ExCamera, a system that can edit, transform, and encode a video, including 4K and VR material, with low latency. The system makes two major contributions.

First, we designed a framework to run general-purpose parallel computations on a commercial "cloud function" service. The system starts up thousands of threads in seconds and manages inter-thread communication.

Second, we implemented a video encoder intended for fine-grained parallelism, using a functional-programming style that allows computation to be split into thousands of tiny tasks without harming compression efficiency. Our design reflects a key insight: the work of video encoding can be divided into fast and slow parts, with the "slow" work done in parallel, and only "fast" work done serially.

1 Introduction

The pace of data analysis and processing has advanced rapidly, enabling new applications over large data sets. Providers use data-parallel frameworks such as MapReduce [8], Hadoop [12], and Spark [32] to analyze a variety of data streams: click logs, user ratings, medical records, sensor histories, error logs, and financial transactions.

Yet video, the largest source of data transiting the Internet [6], has proved one of the most vexing to analyze and manipulate. Users increasingly seek to apply complex computational pipelines to video content. Examples include video editing, scene understanding, object recognition and classification, and compositing. Today, these jobs often take hours, even for a short movie.

There are several reasons that interactive video-processing applications have yet to arrive. First, video jobs take a lot of CPU. In formats like 4K or virtual reality, an hour of video will typically take more than 30 CPU-hours to process. A user who desires results in a few seconds would need to invoke thousands of threads of execution—even assuming the job can be parallelized into thousands of pieces.

Second, existing video encoders do not permit fine-grained parallelism. Video is generally stored in compressed format, but unlike the per-record compression used by data-parallel frameworks [2], video compression relies on temporal correlations among nearby frames. Splitting the video across independent threads prevents exploiting correlations that cross the split, harming compression efficiency. As a result, video-processing systems generally use only coarse-grained parallelism—e.g., one thread per video, or per multi-second chunk of a video—frustrating efforts to process any particular video quickly.

In this paper, we describe ExCamera, a massively parallel, cloud-based video-processing framework that we envision as the backend for interactive video-processing applications. A user might sit in front of a video-editor interface and launch a query: "render this 4K movie, edited as follows, and with scenes containing this actor given a certain color shade." As with any interactive cloud editor (e.g. Google Docs), the goal is to execute the task quickly and make the result immediately accessible online.

ExCamera makes two contributions:

1. A framework that orchestrates general-purpose parallel computations across a "cloud function" service (§ 2). The advent of such services—AWS Lambda, Google Cloud Functions, IBM OpenWhisk, and Azure Functions—permits new models of interactive cloud computing. A cloud function starts in milliseconds and bills usage in fraction-of-a-second increments, unlike a traditional virtual machine, which takes minutes to start and has a minimum billing time of 10 minutes (GCE) or an hour (EC2). Though these services were designed for Web microservices and event handlers, AWS Lambda has additional features that make it more broadly useful: workers can run arbitrary Linux executables and make network connections. ExCamera invokes thousands of C++-implemented Lambda functions in seconds.

2. A video encoder intended for massive fine-grained parallelism (§ 3), built to avoid the traditional compromise of parallel video encoding: the inability to benefit from visual similarities that span parts of the video handled by different threads. ExCamera encodes tiny chunks of the video in independent threads (doing most of the "slow" work in parallel), then stitches those chunks together in a "fast" serial pass, using an encoder written in explicit state-passing style with named intermediate states (§ 4).

Summary of results

We characterized ExCamera's performance and found that it can summon 3,600 cores (representing about 9 teraFLOPS) within 2.5 seconds of job startup. For video encoding, we tuned ExCamera and compared it with existing systems using a 15-minute animated movie in 4K resolution [20], encoding into the VP8 compressed-video format [29]. ExCamera achieved comparable compression to existing systems, at the same quality level relative to the original uncompressed video, and was many times faster. The evaluation is fully described in Section 5.

System	Bitrate at 20 dB SSIM (lower is better)	Encode time (lower is better)
ExCamera[6,16][1]	27.4 Mbps	2.6 minutes
ExCamera[6,1][2]	43.1 Mbps	0.5 minutes
`vpxenc` multi-threaded	27.2 Mbps	149 minutes
`vpxenc` single-threaded	22.0 Mbps	453 minutes
YouTube H.264[3]	*n/a*	36.5 minutes
YouTube VP9	*n/a*	417 minutes

Results were similar on a 12-minute live-action 4K video [23]:

System	Bitrate at 16 dB SSIM (lower is better)	Encode time (lower is better)
ExCamera[6,16]	39.6 Mbps	2.2 minutes
ExCamera[6,1]	66.0 Mbps	0.5 minutes
`vpxenc` multi-threaded	36.6 Mbps	131 minutes
`vpxenc` single-threaded	29.0 Mbps	501 minutes

This paper proceeds as follows. In Section 2, we introduce our framework to execute general-purpose parallel computations, with inter-thread communication, on AWS Lambda. We discuss the problem of fine-grained parallel video encoding in Section 3, and describe ExCamera's approach in Section 4. We then evaluate ExCamera's performance (§ 5), discuss limitations (§ 6), and compare with related work (§ 7).

ExCamera is free software. The source code and evaluation data are available at https://ex.camera.

2 Thousands of tiny threads in the cloud

While cloud services like Amazon EC2 and Google Compute Engine allow users to provision a cluster of powerful machines, doing so is costly: VMs take minutes to start and usage is billed with substantial minimums: an hour for EC2, or 10 minutes for GCE. This means that a cluster of VMs is not effective for running occasional short-lived, massively parallel, interactive jobs.

Recently, cloud providers have begun offering *microservice frameworks* that allow systems builders to replace long-lived servers processing many requests with short-lived workers that are dispatched as requests arrive. As an example, a website might generate thumbnail images using a "cloud function," spawning a short-lived worker each time a customer uploads a photograph.

Because workers begin their task quickly upon spawning and usage is billed at a fine grain, these frameworks show promise as an alternative to a cluster of VMs for short-lived interactive jobs. On the other hand, microservice frameworks are typically built to execute asynchronous lightweight tasks. In contrast, the jobs we target use thousands of simultaneous threads that execute heavyweight computations and communicate with one another.

To address this mismatch we built mu, a library for designing and deploying massively parallel computations on AWS Lambda. We chose AWS Lambda for several reasons: (1) workers spawn quickly, (2) billing is in subsecond increments, (3) a user can run many workers simultaneously, and (4) workers can run arbitrary executables. Other services [4, 10, 19] offer the first three, but to our knowledge none offers the fourth. We therefore restrict our discussion to AWS Lambda. (In the future it may be possible to extend mu to other frameworks.)

In the next sections, we briefly describe AWS Lambda (§ 2.1); discuss the mismatch between Lambda and our requirements, and the architecture that mu uses to bridge this gap (§ 2.2); detail mu's software interface and implementation (§ 2.3); and present microbenchmarks (§ 2.4). We present an end-to-end evaluation of mu applied to massively parallel video encoding in Section 5 and discuss mu's limitations in Section 6.

2.1 AWS Lambda overview

AWS Lambda is a microservice framework designed to execute user-supplied *Lambda functions* in response to asynchronous *events*, e.g., message arrivals, file uploads, or API calls made via HTTP requests. Upon receiving an event, AWS Lambda spawns a *worker*, which executes in a Linux container with configurable resources up to two, 2.8 GHz virtual CPUs, 1,536 MiB RAM, and about 500 MB of disk space. AWS Lambda provisions additional containers as necessary in response to demand.

To create a Lambda function, a user generates a package containing code written in a high-level language (currently Python, Java, Javascript, or C#) and *installs* the package using an HTTP API. Installed Lambda functions are *invoked* by AWS Lambda in response to any of a number of events specified at installation.

[1] "ExCamera[6,16]" refers to encoding chunks of six frames independently in parallel, then stitching them together in strings of 16 chunks.

[2] Encodes independent chunks of six frames each, without stitching.

[3] Because YouTube doesn't encode into the VP8 format at this resolution or expose an adjustable quality, we report only the total time between the end of upload and the video's availability in each format.

At the time of writing, AWS Lambda workers using maximum resources cost 250 μ¢ per 100 ms, or $0.09 per hour. This is slightly less than the closest AWS EC2 instance, c3.large, which has the same CPU configuration, about twice as much RAM, and considerably more storage. While Lambda workers are billed in 100 ms increments, however, EC2 instances are billed hourly; thus, workers are much less expensive for massively parallel computations that are short and infrequent.

2.2 Supercomputing as a (μ)service

With mu, we use AWS Lambda in a different way than intended. Instead of invoking workers in response to a single event, we invoke them in bulk, thousands at a time. The mismatch between our use case and Lambda's design caused several challenges:

1. Lambda functions must be installed before being invoked, and the time to install a function is much longer than the time to invoke it.
2. The timing of worker invocations is unpredictable: workers executing warm (recently invoked) functions spawn more quickly than those running cold functions (§ 2.4). In addition, workers may spawn out of order.
3. Amazon imposes a limit on the number of workers a user may execute concurrently.
4. Workers are behind a Network Address Translator (NAT). They can initiate connections, but cannot accept them, and so they must use NAT-traversal techniques to communicate with one another.
5. Workers are limited to five minutes' execution time.

To illustrate the effect of these limitations, consider a strawman in which a user statically partitions a computation among a number of threads, each running a different computation, then creates and uploads a Lambda function corresponding to each thread's work. First, this requires the user to upload many different Lambda functions, which is slow (limitation 1). Second, workers will spawn slowly, because each one is cold (limitation 2). Third, if there are execution dependencies among the workers, the computation may deadlock if workers are spawned in a pathological order and the number of needed workers exceeds the concurrency limit (limitations 2 and 3). Finally, workers cannot communicate and must synchronize indirectly, e.g., using the AWS S3 block store (limitation 4).

To address issues 1–4, we make three high-level decisions. First, mu uses a long-lived coordinator that provides command and control for a fleet of ephemeral workers that contain no thread-specific logic. Instead, the coordinator steps workers through their tasks by issuing RPC requests and processing the responses. For computations in which workers consume outputs from other workers, mu's coordinator uses dependency-aware scheduling: the coordinator first assigns tasks whose outputs are consumed, then assigns tasks that consume those outputs. This helps to avoid deadlock and reduce end-to-end completion time.

Second, all workers in mu use the same generic Lambda function. This Lambda function is capable of executing the work of any thread in the computation. As described above, at run time the coordinator steps it through its assigned task. This means the user only installs one Lambda function and thus that workers spawn quickly because the function remains warm. Third, we use a *rendezvous server* that helps each worker communicate with other workers. The end result is a highly parallel, distributed, low-latency computational substrate.

This design does not completely sidestep the above limitations. Amazon was willing to increase our concurrent worker limit only to 1,200 per AWS region (the default is 100). We still need to partition our largest computations among several regions. In addition, the five-minute worker timeout seems unavoidable. For ExCamera this limitation does not cause serious problems since the system aims for smaller end-to-end latencies, but it prevented us from benchmarking certain alternative systems (§ 5).

2.3 mu software framework

Workers. mu workers are short-lived Lambda function invocations. When a worker is invoked, it immediately establishes a connection to the coordinator, which thereafter controls the worker via a simple RPC interface. As examples, the coordinator can instruct the worker to retrieve from or upload to AWS S3; establish connections to other workers via a rendezvous server; send data to workers over such connections; or run an executable. For long-running tasks like transferring data and running executables, the coordinator can instruct the worker to run its task in the background.

The user can include additional executables in the mu worker Lambda function package (§ 2.1).[4] The worker executes these in response to RPCs from the coordinator.

Coordinator. The coordinator is a long-lived server (e.g., an EC2 VM) that launches jobs and controls their execution. To launch jobs, the coordinator generates events, one per worker, using AWS Lambda API calls via HTTP (§ 2.1). These HTTP requests are a bottleneck when launching thousands of workers, so the coordinator uses many parallel TCP connections to the HTTP server (one per worker) and submits all events in parallel.

The coordinator contains all of the logic associated with

[4] We statically link these executables to make sure they will run in the worker's environment.

a given computation in the form of per-worker finite-state-machine descriptions. For each worker, the coordinator maintains an open TLS connection, the worker's current state, and its state- transition logic. When the coordinator receives a message from a worker, it applies the state-transition logic to that message, producing a new state and sending the next RPC request to the worker.

Rendezvous. Like the coordinator, the rendezvous server is long lived. mu's rendezvous is a simple relay server that stores messages from workers and forwards them to their destination. This means that the rendezvous server's connection to the workers can be a bottleneck, and thus fast network connectivity between workers and the rendezvous is required. Future work is implementing a hole-punching NAT-traversal strategy to address this limitation.

Developing computations with mu. To design a computation, a user specifies each worker's sequence of RPC requests and responses in the form of a finite-state machine (FSM), which the coordinator executes. mu provides a toolbox of reusable FSM components as a Python library. These components hide the details of the coordinator's communication with workers, allowing a user to describe workers' tasks at a higher level of abstraction.

The simplest of mu's state-machine components represents a single exchange of messages: the coordinator waits for a message from the worker, sends an RPC request, and transitions unconditionally to a new state. For long, straight-line exchanges, mu can automatically pipeline, i.e., send several RPCs at once and then await all of the responses. This is useful if the coordinator's link to workers has high latency.

To encode control-flow constructs like if-then-else and looping, the toolbox includes *state combinator* components. These components do not wait for messages or send RPCs. Instead, they implement logic that encodes conditional FSM transitions. As an example, an if-then-else combinator might check whether a previous RPC succeeded, only uploading a result to S3 upon success. In addition to control flow, mu's state combinators implement cross-worker synchronization and encode parallel execution of multiple RPCs.

Implementation details. mu comprises about 2,700 lines of Python and 2,200 lines of C++. Workers mutually authenticate with the coordinator and rendezvous servers using TLS certificates.

2.4 mu microbenchmarks

In this section, we run microbenchmarks on AWS Lambda using mu to answer two questions:

1. How does cold vs. warm start affect end-to-end latency for massively parallel jobs using mu?
2. How much computational power can mu provide, and how quickly, for an embarrassingly parallel job?

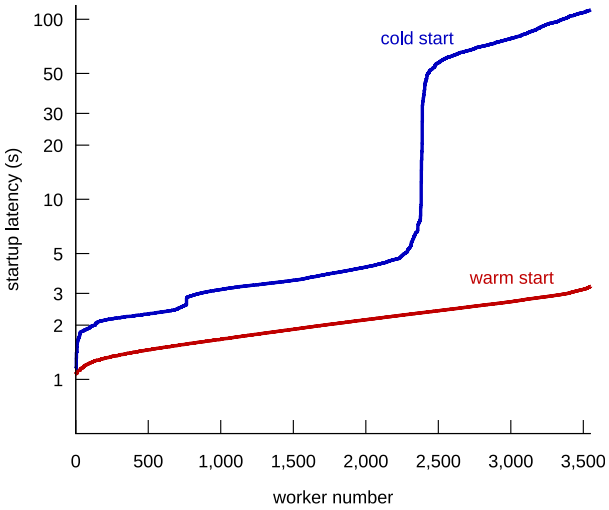


Figure 1: Characterization of cold vs. warm startup time. On a warm start, mu established 3,600 TLS connections to AWS Lambda, spawned 3,600 workers, and received inbound network connections from each worker, in under three seconds (§ 2.4). On a cold start, the delay was as long as two minutes.

In sum, we find that cold starts can introduce minutes of delay, and that a mu job with thousands of threads can run at about nine TFLOPS and starts in just a few seconds when warm. In Section 5, we evaluate mu using end-to-end benchmarks involving worker-to-worker communication and synchronization.

Setup and method. In these microbenchmarks, we invoke 3,600 workers across eight AWS Lambda regions, 450 workers per region. We choose this number because it is similar to the largest computations in our end-to-end evaluation of ExCamera (§ 5). Each worker runs with the maximum allowable resources (§ 2.1). We run a mu job on every worker that executes LINPACK [16], a standard floating-point benchmark that factorizes a random matrix and uses the result to solve a linear equation. Our experiments use double-precision floating point and a $5{,}000 \times 5{,}000$ matrix.

Cold and warm start. Figure 1 shows typical worker startup latency for cold and warm Lambda functions. In both cases, there is about a 1-second delay while the coordinator establishes many parallel TLS connections to the AWS Lambda API endpoint. For cold behavior, we install a new mu worker Lambda function and record the timing on its first execution. Only a few workers start before 2 seconds, and about 2,300 have started by 5 seconds; this delay seems to be the result of Lambda's provisioning additional workers. The last 1,300 cold workers are delayed by more than one minute as a result of rate limiting logic,

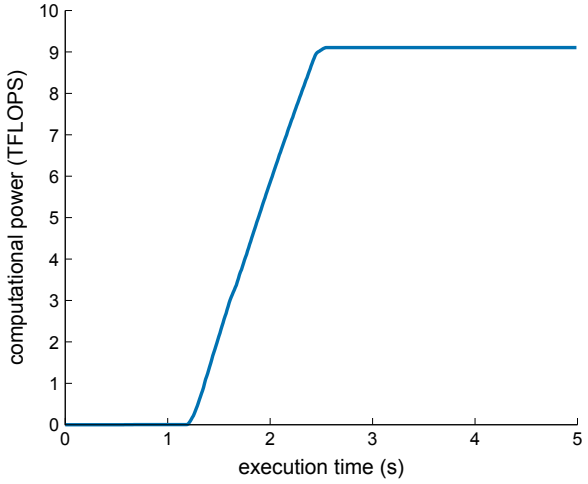


Figure 2: Characterization of floating-point computational power for an embarrassingly parallel job (after PyWren [14]). mu starts 3,600 workers in less than 3 seconds. Each worker executes about 2.5 MFLOPS, for a total of 9 TFLOPS.

which we confirm via the AWS Lambda web interface.

To produce warm behavior, we invoke the same mu computation three times in succession. The result is that AWS Lambda provisions many workers and can start them all very quickly: all 3,600 are started within 3 seconds.

Raw compute power. Figure 2 shows floating-point computational power versus time for a typical (warm) execution of the LINPACK benchmark. All workers start within 3 seconds, and each takes about 3 minutes to run this benchmark. On average, workers execute about 2500 MFLOPS each, for a total of about 9 TFLOPS.

3 Fine-grained parallel video encoding

The previous section described ExCamera's execution engine for running general-purpose parallel jobs with inter-thread communication on a "cloud function" service. We now describe the initial application of this framework: low-latency video encoding to support an interactive application. Parallelizing this workload into tiny tasks was a major focus of our work, because video encoding had not been considered a finely parallelizable operation without compromising compression efficiency.

Compressed video accounts for about 70% of consumer Internet traffic [6]. In digital-video systems, an encoder consumes raw frames and turns them into a compressed bitstream. A corresponding decoder converts the bitstream back to frames that approximate the original input and can be displayed to the viewer.

Video-compression techniques typically exploit the correlations between nearby frames. For example, if a background does not change between two frames, those sections of the image don't need to be repeated in the bitstream. If part of the image is in motion, the encoder can still take advantage of the correlation by using "motion-compensated prediction," which moves around pieces of earlier frames to produce an approximation of the new frame. An encoder spends most of its CPU time searching for such correlations to reduce the size of the output.

These techniques cause compressed frames to depend on one another in a way that makes it impossible to start decoding in midstream. In practice, however, applications often want to start in midstream. For example, a television viewer may change the channel, or a YouTube or Netflix client may want to switch to a higher-quality stream.

To allow this, video encoders insert *Stream Access Points* in the compressed bitstream—one at the beginning, and additional ones in the middle. A Stream Access Point resets the stream and serves as a dependency barrier: the decoder can begin decoding the bitstream at a Stream Access Point, without needing access to earlier portions of the bitstream. The same concept[5] goes by different names in different video formats: a "closed Group of Pictures" (MPEG-2 [17]), an "Instantaneous Display Refresh" (H.264 [11]), or a "key frame" (VP8/VP9 [26, 29]). We use the term "key frame" in this paper.[6]

Chunks of compressed video separated by a key frame are essentially independent. This can be useful in parallel video encoding. Each thread can encode a different range of the video's frames, outputting its own compressed bitstream independently, without making reference to compressed frames in another thread's portion. The resulting bitstreams can simply be concatenated.

But key frames incur a significant cost. As an example, a raw frame of 4K video is about 11 megabytes. After compression in the VP8 format at 15 Mbps, a key frame is typically about one megabyte long. If the video content stays on the same scene without a cut, subsequent frames—those allowed to depend on earlier frames—will be about *10–30 kilobytes*. Spurious insertion of key frames significantly increases the compressed bitrate.

This phenomenon makes low-latency parallel video encoding difficult. To achieve good compression, key frames should only be inserted rarely. For example, systems like YouTube use an interval of four or five seconds. If frames within each chunk are processed serially, then multi-second chunks make it challenging to achieve low-latency pipelines, especially when current encoders process 4K videos at roughly 1/30th (VP8) or 1/100th (VP9) of real time on an x86-64 CPU core. In the next section, we describe how ExCamera addresses this difficulty.

[5] Formally, a Stream Access Point of type 1 or type 2 [18].

[6] An "I-frame" is not the same concept: in MPEG standards, I-pictures depend on compression state left behind by previous elements in the bitstream, and do not prevent subsequent frames from depending on the image content of earlier frames.

4 ExCamera's video encoder

ExCamera is designed to achieve low latency with fine-grained parallelism. Our encoder reflects one key insight: that the work of video encoding can be divided into fast and slow parts, with the "slow" work done in parallel across thousands of tiny threads, and only "fast" work done serially. The slower the per-frame processing—e.g., for pipelines that use 4K or 8K frames or sophisticated video compression, or that run a computer-vision analysis on each frame—the more favorable the case for fine-grained parallelism and for our approach.

ExCamera first runs thousands of parallel `vpxenc` processes (Google's optimized encoder), each charged with encoding just ¼ second, or six frames, of video.[7] Each `vpxenc` output bitstream begins with a key frame. Each thread then re-encodes the first frame, using ExCamera's own video encoder, to remove the key frame and take advantage of correlations with the previous thread's portion of the video. Finally, in a serial step, the system "rebases" each frame on top of the previous one, stitching together chains of 16 chunks. The result is a video that only incurs the penalty of a key frame every four seconds, similar to the interval used by YouTube and other systems.

We refer to this algorithm as "ExCamera[6,16]," meaning it stitches together chains of 16 six-frame chunks. In our evaluation, we profile several settings of these parameters, including ones that simply use naive parallel encoding without stitching (e.g., "ExCamera[6,1]"). ExCamera has been implemented with the VP8 format, but we believe the "rebasing" technique is general enough to work with any recent compressed-video format.

Overview of approach. ExCamera encodes videos using a parallel-serial algorithm with three major phases:

1. *(Parallel)* Each thread runs a production video encoder (`vpxenc`) to encode six compressed frames, starting with a large key frame.

2. *(Parallel)* Each thread runs our own video encoder to replace the initial key frame with one that takes advantage of the similarities with earlier frames.

3. *(Serial)* Each thread "rebases" its chunk of the video onto the prior thread's output, so that the chunks can be played in sequence by an unaltered VP8 decoder without requiring a key frame in between.

In this section, we describe the video-processing primitives we built to enable this algorithm (§ 4.1–4.3) and then specify the algorithm in more detail (§ 4.4).

[7] Major motion pictures are generally shown at 24 frames per second.

4.1 Video in explicit state-passing style

Traditional video encoders and decoders maintain a substantial amount of opaque internal state. The decoder starts decoding a sequence of compressed frames at the first key frame. This resets the decoder's internal state. From there, the decoder produces a sequence of raw images as output. The decoder's state evolves with the stream, saving internal variables and copies of earlier decoded images so that new frames can reference them to exploit their correlations. There is generally no way to import or export that state to resume decoding in midstream.

Encoders also maintain internal state, also with no interface to import or export it. A traditional encoder begins its output with a key frame that initializes the decoder.

For ExCamera, we needed a way for independent encoding threads to produce a small amount of compressed output each, *without* beginning each piece with a key frame. The process relies on a "slow" but parallel phase—compressing each video frame by searching for correlations with recent frames—and a "fast" serial portion, where frames are "rebased" to be playable by a decoder that just finished decoding the previous thread's output.[8]

To allow ExCamera to reason about the evolution of the decoder as it processes the compressed bitstream, we formulated and implemented a VP8 encoder and decoder in explicit state-passing style. A VP8 decoder's state consists of the *probability model*—tables that track which values are more likely to be found in the video and therefore consume fewer bits of output—and three *reference images*, raw images that contain the decoded output of previous compressed frames:

```
state := (prob_model, references[3])
```

Decoding takes a state and a compressed frame, and produces a new state and a raw image for display:

decode(`state`, `compressed_frame`) → (`state′`, `image`)

ExCamera's decode operator is a deterministic pure function; it does not have any implicit state or side effects. The implementation is about 7,100 lines of C++11, plus optimized assembly routines borrowed from Google's `libvpx` where possible. Our implementation passes the VP8 conformance tests.

4.2 What's in a frame?

A compressed frame is a bitstring, representing either a key frame or an "interframe." A key frame resets and

[8] We use the term by analogy to Git, where commits may be written independently in parallel, then "rebased" to achieve a linear history [5]. Each commit is rewritten so as to be applicable to a source-code repository in a state different from when the commit was originally created.

initializes the decoder's state:

decode(*any state*, `key_frame`) → (`state`′, `image`)

An interframe does depend on the decoder's state, because it re-uses portions of the three reference images. An interframe contains a few important parts:

```
interframe :=
  (prediction_modes, motion_vectors, residue)
```

The goal of an interframe is to be as short as possible, by exploiting correlations between the intended output image and the contents of the three reference slots in the decoder's state. It does that by telling the decoder how to assemble a "prediction" for the output image. Every 4x4-pixel square is tagged with a *prediction mode* describing where to find visually similar material: either one of the three references, or elsewhere in the current frame.

For prediction modes that point to one of the three references, the 4x4-pixel square is also tagged with a *motion vector*: a two-dimensional vector, at quarter-pixel precision, that points to a square of the reference image that will serve as the prediction for this square.

When decoding an interframe, the decoder uses the prediction modes and motion vectors to assemble the *motion-compensated prediction image*. It then adds the *residue*—an overlay image that corrects the prediction—applies a smoothing filter, and the result is the output image. The interframe also tells the decoder how to update the probability model, and which reference slots in the state object should be replaced with the new output image.

The goal is for the prediction to be as accurate as possible so the residue can be coded in as few bits as possible. The encoder spends most of its time searching for the best combination of prediction modes and motion vectors.

4.3 Encoding and rebasing

ExCamera's main insight is that the hard part of encoding an interframe—finding the best prediction modes and motion vectors—can be run in parallel, while the remaining work of calculating the residue is fast enough to be serialized. This enables most of the work to be parallelized into tiny chunks, without requiring a large key frame at the start of each thread's output.

To do this, we had to implement two more VP8 operations, in about 3,400 lines of C++11. The first is a video encoder that can start from an arbitrary decoder state:

encode-given-state(`state`, `image`, *quality*) → `interframe`

This routine takes a state (including the three reference images), and a raw input image, and searches for the best combination of motion vectors and prediction modes so that the resulting interframe approximates the original image to a given fidelity.

Our encoder is not as good as Google's `vpxenc`, which uses optimized vector assembly to search for the best motion vectors. Compared with `vpxenc`, "encode-given-state" is much slower and produces larger output for the same quality (meaning, similarity of the decoder's output image to the input image). However, it can encode given an externally supplied state, which allows the key frame at the start of each chunk to be replaced with an interframe that depends on the previous thread's portion of the video.

The second operation is a "rebase" that transforms interframes so they can be applied to a different state than the one they were originally encoded for:

rebase(`state`, `image`, `interframe`) → `interframe`′

While encode-given-state creates a compressed frame *de novo*, a rebase is a transformation on compressed frames, taking advantage of calculations already done. Rebasing involves three steps:

1. **Don't redo the slow part.** Rebasing adopts verbatim the prediction modes and motion vectors from the original interframe.
2. **Apply motion prediction to new state.** Rebasing applies those prediction modes and motion vectors to the *new* `state` object, producing a new motion-compensated prediction image.
3. **Recalculate residue given original target image.** Rebasing subtracts the motion-compensated prediction image from the *original* input to the encoder: the raw target image that was taken "ex camera," i.e., directly from the camera. This "fast" subtraction produces a new residue, which is encoded into the output `interframe`′.

4.4 The parallel-serial algorithm

We now describe an ExCamera[N, x] encoder pipeline. The algorithm works separately on batches of x threads, each handling N frames, so the entire batch accounts for $N \cdot x$ frames and will contain only one key frame. For example, the ExCamera[6,16] algorithm uses batches of 96 frames each (meaning there will be one key frame every four seconds). Batches are run in parallel. Within each batch, the x threads proceed as follows:

1. *(Parallel)* Each thread downloads an N-image chunk of raw video. At the resolution of a 4K widescreen movie, each image is 11 megabytes.
2. *(Parallel)* Each thread runs Google's `vpxenc` VP8 encoder. The output is N compressed frames: one key frame (typically about one megabyte) followed by $N-1$ interframes (about 10–30 kilobytes apiece).

3. *(Parallel)* Each thread runs ExCamera's **decode** operator N times to calculate the final state, then sends that state to the next thread in the batch.

4. *(Parallel)* The first thread is now finished and uploads its output, starting with a key frame. The other $x-1$ threads run **encode-given-state** to encode the first image as an interframe, given the state received from the previous thread. The key frame from `vpxenc` is thrown away; **encode-given-state** works *de novo* from the original raw image.[9]

5. *(Serial)* The first remaining thread runs **rebase** to rewrite interframes 2..N in terms of the state left behind by its new first frame. It sends its final state to the next thread, which runs **rebase** to rewrite all its frames in terms of the given state. Each thread continues in turn. After a thread completes, it uploads its transformed output and quits.

The basic assumption of this process is that rebasing (recalculating residues by subtracting the prediction from the intended image) can be done quickly, while the difficult work of searching for correlations, and therefore motion vectors, can be done in parallel.

◰ (original) ◳ SSIM = 14.3 dB
◱ SSIM = 11.4 dB ◲ SSIM = 9.4 dB

Figure 3: Structural similarity (SSIM) [28] correlates with perceived image quality.

5 Evaluation

We evaluated the ExCamera encoder, running on AWS Lambda in the mu framework, on four metrics: (1) job completion time, (2) bitrate of the compressed output, (3) quality of the output, compared with the original, (4) monetary cost of execution.

In summary, ExCamera achieved similar bitrates and quality as a state-of-the-art video encoder—Google's `vpxenc` running with multi-threading on a 128-core machine—while running about 60× faster. Pipelines with fine-grained parallelism and rebasing (e.g. ExCamera[6,16]) achieved similar results to pipelines with coarser-grained parallelism and no rebasing (e.g. ExCamera[24,1]). Our tests were performed on two open-source movies that are available in uncompressed 4K raw format: the 15-minute animated "Sintel" (2010) [20], and the 12-minute live-action "Tears of Steel" (2012) [23].

5.1 Methods and metrics

To set up the evaluation, we built and executed a mu pipeline (§ 2) to convert the raw frames of Sintel—distributed as individual PNG files—into chunks of raw images in the yuv4mpeg 8-bit 4:2:0 format.[10] Each chunk was uploaded to Amazon S3. To stay under Amazon's limit of 1,200 concurrent workers per AWS region, we spread the workers and the movies across four regions.

Our evaluation uses two other mu pipelines. The first implements the ExCamera[N,x] video-encoding algorithm described in § 4.4. We partitioned the encoding job into four large pieces, each accounting for a quarter of the movie, and ran one piece in each region. Within a region, the algorithm runs x-thread batches independently in parallel. Each batch produces $N \cdot x$ frames of compressed video, the first of which is a key frame.

Within each batch, the mu coordinator assigns tasks with awareness of the serial data-dependency relationships of the encoding algorithm: each thread depends on state objects calculated by the previous thread in the batch. Thus, as workers spawn, the coordinator assigns tasks to batches in a round-robin fashion: all of the first threads across all batches, then all of the second threads, etc. This maximizes the likelihood that threads will make continuous forward progress, avoiding stalls.

The final mu pipeline collects additional performance metrics, defined as follows:

Latency: Total time-to-completion, starting when the coordinator is launched, and ending when all threads have finished uploading compressed output to S3.

Bitrate: Total compressed size divided by duration.

Quality: Fidelity of the compressed frames to the original raw frames. We use the quality metric of *struc-*

[9] We do use the quality, or quantization, settings chosen by `vpxenc` for key frames and interframes to select the quality of the new interframe.

[10] "Tears of Steel" was already available in this format.

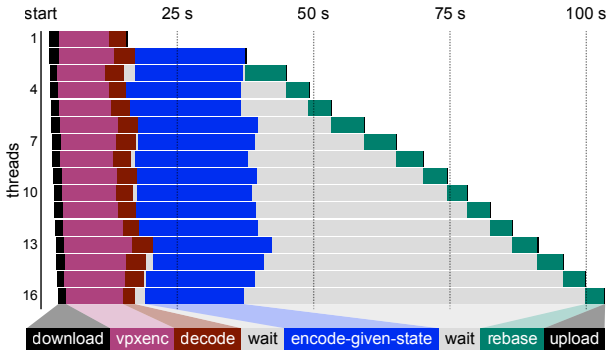


Figure 4: Execution of the ExCamera[6,16] encoder (§ 4.4) for a typical 16-thread batch on a warm start. At two points in the computation, a thread may have to "wait" for state from the thread above. The "slow" work of searching for motion vectors and prediction modes (`vpxenc` and encode-given-state) runs in parallel. The "fast" rebasing step runs serially.

tural similarity (SSIM) [28], calculated by the Xiph `dump_ssim` tool [7]. Figure 3 illustrates how SSIM correlates with perceived image quality.

5.2 Baselines

We benchmarked a range of ExCamera[N,x] pipelines against two alternative VP8 encoding systems:

1. **vpx (single-threaded):** `vpxenc`, running in single-threaded mode on an Amazon EC2 c3.8xlarge instance with a 2.8 GHz Intel Xeon E5-2680 v2 CPU. The source material was on a local SSD RAID array. Encoded output was written to a RAM disk.

   This represents the best possible compression and quality that can be achieved with a state-of-the-art encoder, but takes about 7.5 hours to encode Sintel.

2. **vpx (multi-threaded):** `vpxenc` in multi-threaded mode and configured to use all cores, running on the same kind of instance as above. This represents a common encoding configuration, with reasonable tradeoffs among compression, quality, and speed. Encoding Sintel takes about 2.5 hours. We also ran `vpxenc` on a 128-core x1.32xlarge machine, with the same results. `vpxenc`'s style of multi-threading parallelizes compression *within* each frame, which limits the granularity of parallelism.

5.3 Results

Microbenchmark. We first present a breakdown of the time spent by an ExCamera[6,16] encoder. This serves as a microbenchmark of the mu system and of the algorithm. Figure 4 shows the results of a typical batch. mu's dependency-aware scheduler assigns the first worker that AWS Lambda spawns to a task whose output will be consumed by later-spawning workers. The figure shows that the total amount of "slow" work dwarfs the time spent on rebasing, but the serial nature of the rebasing step makes it account for most of the end-to-end completion time.

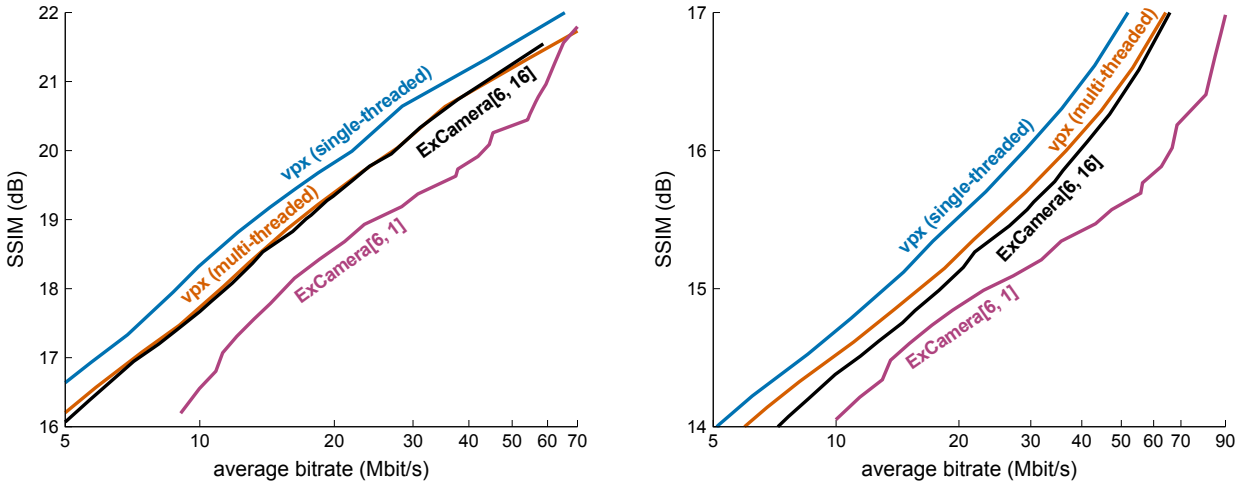
The analysis confirms that our own encode routine (encode-given-state) is considerably slower than Google's `vpxenc`, especially as the latter is encoding six frames and our routine only one. This suggests ExCamera has considerable room for optimization remaining.

Encoder performance. We ran ExCamera and the baselines with a variety of encoding parameters, resulting in a range of quality-to-bitrate tradeoffs for each approach. Figure 5 shows the results. As expected, vpx (single-threaded) gives the best quality at a given bitrate, and naive parallelism (ExCamera[6,1]) produces the worst. On the animated movie, ExCamera[6,16] performs as well as multi-threaded `vpxenc`, giving within 2% of the same quality-to-bitrate tradeoff with a much higher degree of parallelism. On the live-action movie, ExCamera is within 9%.

We also measured each approach's speed. We chose a particular quality level: for Sintel, SSIM of 20 dB, representing high quality, and for Tears of Steel, SSIM of 16 dB. We ran ExCamera with a range of settings that produce this quality, as well as the baselines, to compare the tradeoffs of encoding speed and compressed bitrate. We took the median of at least three runs of each scheme and linearly interpolated time and bitrate between runs at adjacent quality settings when we could not achieve exactly the target quality.

Figure 6 shows the results. ExCamera sweeps out a range of tradeoffs between 60× to 300× faster than multi-threaded `vpxenc`, with compression that ranges between 10% better and 80% worse. In some cases, pipelines with coarser-grained parallelism and no rebasing (e.g. ExCamera[24,1]) outperformed pipelines with finer-grained parallelism and rebasing (e.g. ExCamera[6,16]). This suggests inefficiency in our current implementation of **encode-given-state** and **rebase** that can be improved upon, but at present, the value of fine-grained parallelism and rebasing may depend on whether the application pipeline includes other costly per-frame processing, such as a filter or classifier, in addition to compressing the output. The costlier the per-frame computation, the more worthwhile it will be to use fine-grained threads.

YouTube measurement. To compare against a commercial parallel encoding system, we uploaded the official H.264 version of Sintel, which is 5.1 GiB, to YouTube. YouTube appears to insert key frames every 128 frames, and we understand that YouTube parallelizes at least some encoding jobs with the same granularity. The upload took 77 seconds over a gigabit Ethernet connection from



(a) Sintel (animated)

(b) Tears of Steel (live-action)

Figure 5: Quality vs. bitrate for four VP8 video encoders on the two 4K movies. ExCamera[6,16] achieves within 2% (Sintel) or 9% (Tears of Steel) of the performance of a state-of-the-art encoder (multi-threaded `vpxenc`) with a much higher degree of parallelism.

Stanford. We ran `youtube-dl --list-formats` every five seconds to monitor the availability of the processed versions. Counting from the end of the upload, it took YouTube 36.5 minutes until a compressed H.264 version was available for playback. It took 417 minutes until a compressed VP9 version was available for playback.

Because YouTube does not encode 4K VP8 content, and does not have adjustable quality, these figures cannot be directly compared with those in Figure 6. However, they suggest that even in systems that have no shortage of raw CPU resources, the coarse granularity of available parallelism may be limiting the end-to-end latency of user-visible encoding jobs.

Cost. At AWS Lambda's current pricing, it costs about $5.40 to encode the 15-minute Sintel movie using the ExCamera[6,16] encoder. The encoder runs 3,552 threads, each processing ¼ second of the movie. The last thread completes after 2.6 minutes, but because workers quit as soon as their chunk has been rebased and uploaded, the average worker takes only 60.4 seconds. In a long chain of rebasing, later threads spend much of their time waiting on predecessors (Figure 4). A more-sophisticated launching strategy could save money, without compromising completion time, by delaying launching these threads.

6 Limitations and future work

At present, ExCamera has a number of limitations in its evaluation, implementation, and approach. We discuss these in turn.

6.1 Limitations of the evaluation

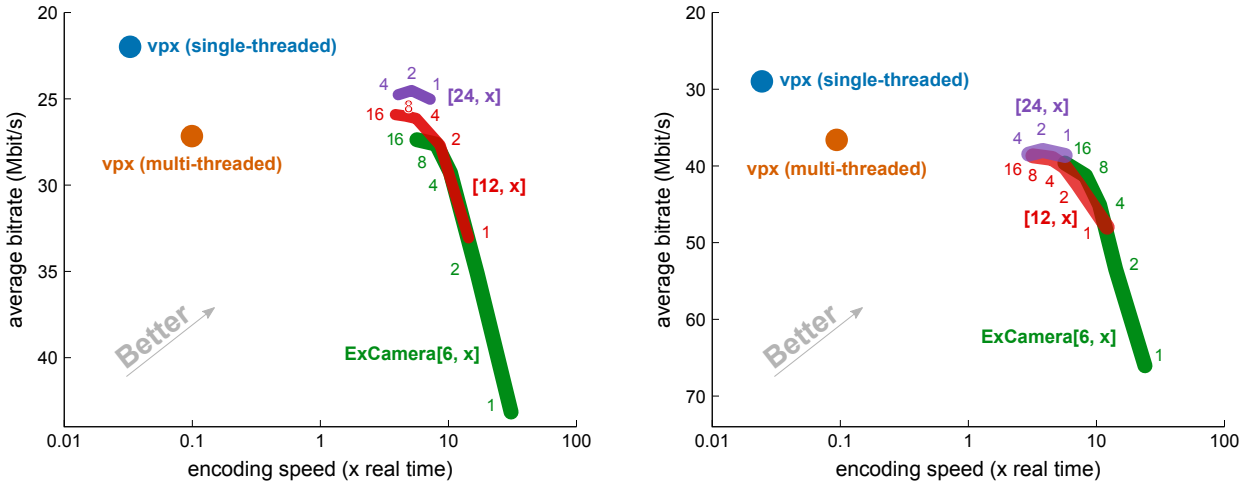
Only evaluated on two videos. We have only characterized ExCamera's performance on two creative-commons videos (one animated, one live-action). While these are widely used benchmarks for video encoders, this may have more to do with their availability in uncompressed formats than suitability as benchmarks. We will need to verify experimentally that ExCamera's results generalize.

If everybody used Lambda as we do, would it still be as good? To the best of our knowledge, ExCamera is among the first systems to use AWS Lambda as a supercomputer-by-the-second. ExCamera slams the system with thousands of TLS connections and threads starting at once, a workload we expect not to be characteristic of other customers. We don't know if Lambda would continue to provide low latencies, and maintain its current pricing, if ExCamera-like workloads become popular.

6.2 Limitations of the implementation

encode-given-state is slow and has poor compression efficiency. The microbenchmark of Figure 4 and other measurements suggest there is considerable room for optimization in our encode-given-state routine (§ 4.4). This is future work. We explored building further on `vpxenc` so as to avoid using our own *de novo* encoder at all, but did not achieve an improvement over the status quo.

Pipeline specification is complex. In addition to parallel video compression, ExCamera supports, in principle, a range of pipeline topologies: per-image transformations followed by encoding, operations that compose multiple input frames, edits that rearrange frames, and computer-



(a) Sintel (SSIM of 20 dB)

(b) Tears of Steel (SSIM of 16 dB)

Figure 6: Bitrate vs. encoding speed, at constant quality relative to the original movie. ExCamera sweeps out a range of tradeoffs.

vision analyses on each frame. We are working to design a pipeline-description language to specify jobs at a higher level of abstraction.

Worker failure kills entire job. Because ExCamera's jobs run for only a few minutes, and the mu framework only assigns tasks to Lambda workers that successfully start up and make a TLS connection back to the coordinator, mu does not yet support recovery from failure of a Lambda worker in the *middle* of a task. Producing Figure 6 required 640 jobs, using 520,000 workers in total, each run for about a minute on average. Three jobs failed.

ExCamera encodes VP8. What about newer formats? It took considerable effort to implement a VP8 decoder, encoder, and rebase routine in explicit state-passing style. ExCamera only works with this format, which was designed in 2008 and has been largely superseded by the newer VP9. Although Google released a VP9 version of `vpxenc` in 2013, the VP9 specification was not released until April 2016. To the best of our knowledge, there has not yet been an independent implementation of VP9.[11] We believe the rebasing technique is general enough to work with VP9 and other current formats, but hesitate to predict the efficacy without an empirical evaluation. We hope to persuade implementers of the benefit of ExCamera's explicit-state-passing style and to provide a similar interface themselves.

6.3 Limitations of the approach

Many video jobs don't require fine-grained parallelism. ExCamera is focused on video pipelines where the processing of a single frame consumes a large amount of CPU resources—e.g., encoding of large (4K or VR) frames, expensive computer-vision analyses or filtering, etc. Only in these cases is it worthwhile to parallelize processing at granularities finer than the practical key-frame interval of a few seconds. For encoding "easier" content (e.g., standard-definition resolutions) without extensive per-frame computation, it is already possible to run a naively parallel approach on multi-second chunks with acceptable end-to-end latency. Figure 6 shows that if the application is simply encoding video into the VP8 format, with no expensive per-frame processing that would benefit from fine-grained parallelism, pipelines with coarser granularity and no rebasing (e.g. ExCamera[24,1]) can sometimes perform as well as pipelines with finer granularity and rebasing (e.g., ExCamera[6,16]).

Assumes video is already in the cloud. We assume that the source material for a video pipeline is already in the cloud and accessible to the workers—in our cases, loaded into S3, either in raw format (as we evaluated) or compressed into small chunks. Many video pipelines must deal with video uploaded over unreliable or challenged network paths, or with compressed source video that is only available with infrequent Stream Access Points.

7 Related work

Data-processing frameworks. Batch-processing frameworks such as MapReduce [8], Hadoop [12], and Spark [32] are suited to tasks with coarse-grained parallelism, such as mining error logs for abnormal patterns. In these tasks, each thread processes an independent subset of the data. It is challenging to express video encoding

[11] Existing software implementations have been written by Google employees or by former employees who worked on `vpxenc`.

within these frameworks, because video compression depends on exploiting the correlations among frames.

A number of additional distributed-computing frameworks have been proposed that implement pipeline-oriented computation. Apache Tez [24] and Dryad [13] support arbitrary DAG-structured computation, with data passed along edges in a computation graph. Such systems could be used to support video-processing pipelines.

Cloud computing. Data-processing frameworks today rely on computational substrates such as Amazon EC2 and Microsoft Azure, which provide heavyweight virtualization through virtual machines that run an entire operating system. In contrast, ExCamera relies on AWS Lambda's cloud functions, which provides lightweight virtualization. For the same monetary cost, it can access many more parallel resources, and can start and stop them faster. To our knowledge, ExCamera is the first system to use such cloud functions for compute-heavy tasks, such as video encoding; current uses [25] are focused on server-side scripts for Web microservices and asynchronous event handlers.

After the submission of this paper, we sent a preprint to a colleague who then developed PyWren [14, 15], a framework that executes thousands of Python threads on AWS Lambda. Our Figure 2 was inspired by a similar measurement in [14]. ExCamera's mu framework differs from PyWren in its focus on heavyweight computation with C++-implemented Linux threads and inter-thread communication.

Parallel video encoding. Parallel video encoding has a substantial literature. Broadly speaking, existing approaches use one of two techniques: intra-frame parallelism, where multiple threads operate on disjoint areas of the same frame [3, 27, 31], or frame parallelism, where each thread handles a different range of frames [1, 22, 30].

Intra-frame parallelism is widely used to enable real-time encoding and decoding with small numbers of threads. This kind of parallelism does not scale with the length of a video, and in practice cannot be increased beyond tens of threads without severely compromising compression efficiency. This limits the speedup available.

Frame parallelism can scale with the length of a videos, but also sacrifices coding efficiency at higher degrees of parallelism (§ 3). Some prior systems [9] employ frame parallelism by searching for natural locations to place key frames, then encoding the ranges between pairs of key frames independently. These systems operate at coarse granularity: thousands of frames per worker. The goal of ExCamera's encoder is to exploit fine-grained parallelism without sacrificing coding efficiency.

Reusing prior encoding decisions. ExCamera's frame-rebasing technique involves transforming an interframe into a new interframe, preserving the motion vectors and prediction modes from an earlier encoding, but recalculating the residue so that the frame can be applied to a new decoder state.

The idea of preserving motion vectors and prediction modes to save CPU is not new: similar techniques have a long history in the context of real-time transcoding (e.g., [21]), where one input stream of compressed video is turned into several output streams at varying bitrates. In these systems, the original motion vectors and prediction modes can be reused across each of the output streams. However, to the best we have been able to determine, ExCamera is the first system to use this idea to enable low-latency parallel video encoding.

8 Conclusion

ExCamera is a cloud-based video-processing framework that we envision as the backend for interactive video applications. It can edit, transform, and encode a video, including 4K and VR material, with low latency.

The system makes two major contributions: a framework to run general-purpose parallel computations on a commercial "cloud function" service with low latency, and a video encoder built with this framework that achieves fine-grained parallelism without harming compression efficiency.

ExCamera suggests that an explicit state-passing style, which exposes the internal state of a video encoder and decoder, is a useful interface that can enable substantial gains for video-processing workloads—applications that will only grow in importance. We encourage the developers of video codecs to implement such abstractions.

9 Acknowledgments

We thank the NSDI reviewers and our shepherd, Dave Maltz, for their helpful comments and suggestions. We are grateful to Timothy Terriberry for feedback throughout this project, and to Philip Levis and Fraser Brown for comments on versions of this paper. We benefited from helpful conversations with Marc Brooker, Tim Wagner, and Peter Vosshall of Amazon, and with James Bankoski, Josh Bailey, Danner Stodolsky, Roberto Peon, and the video-encoding team at Google. This work was supported in part by NSF grants CNS-1528197 and CNS-1564185, a Google Research Award, and an Amazon Cloud Credits award, with additional funding from Dropbox, VMware, and Facebook.

References

[1] AARON, A., AND RONCA, D. High quality video encoding at scale. In *Netflix Tech Blog* (Decem-

ber 9, 2015). http://techblog.netflix.com/2015/12/high-quality-video-encoding-at-scale.html.

[2] AGARWAL, R., KHANDELWAL, A., AND STOICA, I. Succinct: Enabling queries on compressed data. In *NSDI* (May 2015).

[3] AKRAMULLAH, S., AHMAD, I., AND LIOU, M. A data-parallel approach for real-time MPEG-2 video encoding. *Journal of Parallel and Distributed Computing 30*, 2 (1995), 129–146.

[4] Azure Functions—Serverless Architecture. https://azure.microsoft.com/en-us/services/functions/.

[5] CHACON, S., AND STRAUB, B. *Pro Git: Git Branching - Rebasing*, 2nd ed. Apress, Berkeley, CA, USA, 2014. https://git-scm.com/book/en/v2/Git-Branching-Rebasing.

[6] Cisco VNI forecast and methodology, 2015–2020. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html.

[7] Xiph Daala repository. https://github.com/xiph/daala.

[8] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[9] ELDER. http://web.archive.org/web/20080105113156/http://www.funknmary.de/bergdichter/projekte/index.php?page=ELDER.

[10] Cloud Functions—Serverless Microservices. https://cloud.google.com/functions/.

[11] *Advanced video coding for generic audiovisual services*, May 2003. Rec. ITU-T H.264 and ISO/IEC 14496-10 (http://www.itu.int/rec/T-REC-H.264-201602-I/en).

[12] Apache Hadoop. http://hadoop.apache.org.

[13] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys* (Mar. 2007).

[14] JONAS, E. Microservices and teraflops. *http://ericjonas.com/pywren.html* (Oct. 2016).

[15] JONAS, E., VENKATARAMAN, S., STOICA, I., AND RECHT, B. Occupy the Cloud: Distributed computing for the 99%. *https://arxiv.org/abs/1702.04024* (Feb. 2017).

[16] LINPACK_BENCH: The LINPACK benchmark. https://people.sc.fsu.edu/~jburkardt/cpp_src/linpack_bench/linpack_bench.html.

[17] *Generic coding of moving pictures and associated audio information [MPEG-2] – Part 2: Video*, May 1996. Rec. ITU-T H.262 and ISO/IEC 13818-2 (http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=61152).

[18] *Dynamic adaptive streaming over HTTP (DASH) — Part 1: Media presentation description and segment formats*, April 2012. ISO/IEC 23009-1 (http://standards.iso.org/ittf/PubliclyAvailableStandards).

[19] IBM Bluemix OpenWhisk. http://www.ibm.com/cloud-computing/bluemix/openwhisk/.

[20] ROOSENDAAL, T. Sintel. In *SIGGRAPH* (Aug. 2011).

[21] SAMBE, Y., WATANABE, S., YU, D., NAKAMURA, T., AND WAKAMIYA, N. High-speed distributed video transcoding for multiple rates and formats. *IEICE - Trans. Inf. Syst. E88-D*, 8 (Aug. 2005), 1923–1931.

[22] SHEN, K., ROWE, L. A., AND DELP III, E. J. Parallel implementation of an MPEG-1 encoder: faster than real time. In *SPIE* (July 1995).

[23] Tears of Steel: Mango open movie project. https://mango.blender.org.

[24] Apache Tez. https://tez.apache.org/.

[25] Examples of how to use AWS Lambda. http://docs.aws.amazon.com/lambda/latest/dg/use-cases.html.

[26] *VP9 Bitstream & Decoding Process Specification Version 0.6*, March 2016. http://www.webmproject.org/vp9/.

[27] VP8 Encode Parameter Guide. http://www.webmproject.org/docs/encoder-parameters/.

[28] WANG, Z., BOVIK, A. C., SHEIKH, H. R., AND SIMONCELLI, E. P. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing 13*, 4 (2004), 600–612.

[29] WILKINS, P., XU, Y., QUILLIO, L., BANKOSKI, J., SALONEN, J., AND KOLESZAR, J. VP8 Data Format and Decoding Guide. RFC 6386, Nov. 2011.

[30] x264 Settings. http://www.chaneru.com/Roku/HLS/X264_Settings.htm#threads.

[31] YU, Y., AND ANASTASSIOU, D. Software implementation of MPEG-II video encoding using socket programming in LAN. In *SPIE* (Feb. 1994).

[32] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI* (Apr. 2012).

Appendix: `vpxenc` command lines

We used `vpxenc` 1.6.0 (Sintel) and 1.6.1 (Tears of Steel) with the below command-line arguments in our evaluations (§ 5). `$QUALITY` is a parameter that indicates the target quality of the compressed video; it ranges from 0 (best) to 63 (worst). When running vpx multi-threaded, `$NPROC` is set to 31 (one fewer than the number of cores, as recommended in the documentation) and `$TOK_PARTS` is set to 3. When running single-threaded, they are set to 1 and 0, respectively.

```
vpxenc --codec=vp8 \
       --good \
       --cpu-used=0 \
       --end-usage=cq \
       --min-q=0 \
       --max-q=63 \
       --buf-initial-sz=10000 \
       --buf-optimal-sz=20000 \
       --buf-sz=40000 \
       --undershoot-pct=100 \
       --passes=2 \
       --auto-alt-ref=1 \
       --tune=ssim \
       --target-bitrate=4294967295 \
       --cq-level=$QUALITY \
       --threads=$NTHREADS \
       --token-parts=$TOK_PARTS \
       -o output_file \
       input.y4m
```

Live Video Analytics at Scale with Approximation and Delay-Tolerance

Haoyu Zhang[⋆†], Ganesh Ananthanarayanan[⋆], Peter Bodik[⋆], Matthai Philipose[⋆]
Paramvir Bahl[⋆], Michael J. Freedman[†]

[⋆]*Microsoft* [†]*Princeton University*

Abstract

Video cameras are pervasively deployed for security and smart city scenarios, with millions of them in large cities worldwide. Achieving the potential of these cameras requires efficiently analyzing the *live* videos *in real-time*. We describe VideoStorm, a video analytics system that processes thousands of video analytics *queries* on live video streams over large clusters. Given the high costs of vision processing, resource management is crucial. We consider two key characteristics of video analytics: *resource-quality tradeoff with multi-dimensional configurations*, and *variety in quality and lag goals*. VideoStorm's offline profiler generates query resource-quality profile, while its online scheduler allocates resources to queries to *maximize performance* on quality and lag, in contrast to the commonly used fair sharing of resources in clusters. Deployment on an Azure cluster of 101 machines shows improvement by as much as 80% in quality of real-world queries and 7× better lag, processing video from operational traffic cameras.

1 Introduction

Video cameras are pervasive; major cities worldwide like New York City, London, and Beijing have millions of cameras deployed [8, 12]. Cameras are installed in buildings for surveillance and business intelligence, while those deployed on streets are for traffic control and crime prevention. Key to achieving the potential of these cameras is effectively analyzing the *live* video streams.

Organizations that deploy these cameras—cities or police departments—operate large clusters to analyze the video streams [5, 9]. Sufficient bandwidth is provisioned (fiber drops or cellular) between the cameras and the cluster to ingest video streams. Some analytics need to run for long periods (e.g., counting cars to control traffic light durations) while others for short bursts of time (e.g., reading the license plates for AMBER Alerts, which are raised in U.S. cities to identify child abductors [1]).

Video analytics can have *very high resource demands*. Tracking objects in video is a core primitive for many scenarios, but the best tracker [69] in the VOT Challenge 2015 [59] processes only 1 frame per second on an 8-core machine. Some of the most accurate Deep Neural Networks for object recognition, another core primitive, require 30GFlops to process a single frame [75]. Due to the high processing costs and high data-rates of video streams, resource management of *video analytics queries* is crucial. We highlight two properties of video analytics queries relevant to resource management.

Resource-quality trade-off with multi-dimensional configurations. Vision algorithms typically contain various parameters, or *knobs*. Examples of knobs are video resolution, frame rate, and internal algorithmic parameters, such as the size of the sliding window to search for objects in object detectors. A combination of the knob values is a query *configuration*. The configuration space grows exponentially with the number of knobs. *Resource demand* can be reduced by changing configurations (e.g., changing the resolution and sliding window size) but they typically also lower the output *quality*.

Variety in quality and lag goals. While many queries require producing results in real-time, others can tolerate *lag* of even many minutes. This allows for temporarily reallocating some resources from the lag-tolerant queries during interim shortage of resources. Such shortage happens due to a burst of new video queries or "spikes" in resource usage of existing queries (for example, due to an increase in number of cars to track on the road).

Indeed, video analytics queries have a wide variety of quality and lag goals. A query counting cars to control the traffic lights can work with moderate quality (approximate car counts) but will need them with low lag. License plate readers at toll routes [16, 17], on the other hand, require high quality (accuracy) but can tolerate lag of even many minutes because the billing can be delayed. However, license plate readers when used for AMBER Alerts require high quality results *without* lag.

Scheduling large number of streaming video queries with diverse quality and lag goals, each with many configurations, is computationally complex. Production systems for stream processing like Storm [4], StreamScope [62], Flink [2], Trill [36] and Spark Streaming [89] allocate resources among multiple queries only based on *resource fairness* [7, 10, 27, 43, 51] common to cluster managers like Yarn [3] and Mesos [49]. While simple, being agnostic to query quality and lag makes fair sharing far from ideal for video stream analytics.

We present VideoStorm, a video analytics system that scales to processing thousands of *live* video streams over large clusters. Users submit video analytics queries containing many *transforms* that perform vision signal processing on the frames of the incoming video. At its core, VideoStorm contains a scheduler that efficiently generates the query's *resource-quality profile* for its different knob configurations, and then *jointly maximizes the quality and minimizes the lag* of streaming video queries. In doing so, it uses the generated profiles, and lag and quality goals. It allocates resources to each query and picks its configuration (knob values) based on the allocation.

Challenges and Solution. The major technical challenges for designing VideoStorm can be summarized as follows: *(i)* There are no analytical models for resource demand and quality for a query configuration, and the large number of configurations makes it expensive to even estimate the resource-quality profile. *(ii)* Expressing quality and lag goals of *individual queries* and *across all queries* in a cluster is non-trivial. *(iii)* Deciding allocations and configurations is a computationally hard problem *exponential* in the number of queries and knobs.

To deal with the multitude of knobs in video queries, we split our solution into *offline* (or profiling) and *online* phases. In the offline phase, we use an efficient *profiler* to get the resource-quality profile of queries without exploring the entire combinatorial space of configurations. Using greedy search and domain-specific sampling, we identify a handful of knob configurations on the *Pareto boundary* of the profile. The scheduler in the online phase, thus, has to consider only these configurations.

We encode quality and lag goals of a query in a *utility function*. Utility is a weighted combination of the achieved quality and lag, with penalties for violating the goals. Penalties allow for expressing priorities between queries. Given utilities of multiple queries, we schedule for two natural objectives – maximize the *minimum* utility, or maximize the *total* utility. The former achieves fairness (max-min) while the latter targets performance.

Finally, in the online phase, we model the scheduling problem using the Model-Predictive Control [67] to predict the *future* query lag over a short time horizon, and use this predicted lag in the utility function. The scheduler considers the resource-quality profile of queries during allocation, and allows for lagging queries to "catch up." It also deals with inevitable inaccuracies in resource usages in the resource-quality profiles.

While we focus VideoStorm on video analytics using computer vision algorithms, approximation and lag are aspects that are fundamental to all machine learning algorithms. To that end, the techniques in our system are broadly applicable to all stream analytics systems that employ machine learning techniques.

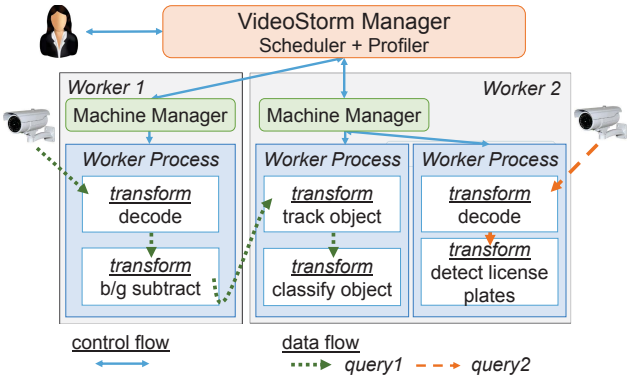


Figure 1: VideoStorm **System Architecture.**

Contributions. Our contributions are as follows:

1. We designed and built a system for large-scale analytics of live video that allows users to submit queries with arbitrary vision processors.
2. We efficiently identify the resource-quality profile of video queries without exhaustively exploring the combinatorial space of knob configurations.
3. We designed an efficient scheduler for video queries that considers their resource-quality profile and lag tolerance, and trades off between them.

We considered streaming databases with approximation [19,37,68] as a starting point for our solution. However, they only consider the sampling rate of data streams and used established analytical models [38] to calculate the quality and resource demand. In contrast, vision queries are more complex black-boxes with many more knobs, and do not have known analytical models. Moreover, they optimize only one query at a time, while our focus is on scheduling multiple concurrent queries.

Deployment on 101 machines in Azure show that VideoStorm's scheduler allocates resources in hundreds of milliseconds even with thousands of queries. We evaluated using real video analytics queries over video datasets from live traffic cameras from several large cities. Our offline profiling consumes 3.5× less CPU resources compared to a basic greedy search. The online VideoStorm scheduler outperforms fair scheduling of resources [3, 31, 49] by as much as 80% in quality of queries and 7× in terms of lag.

2 System Description

We describe the high-level architecture of VideoStorm and the specifications for video queries.

2.1 VideoStorm Architecture

The VideoStorm cluster consists of a centralized *manager* and a set of *worker machines* that execute *queries*,

```
"name": "LicensePlate",
"transforms": [
  {"id": "0",
   "class_name": "Decoder",
   "parameters": {
     "CameraIP": "134.53.8.8",
     "CameraPort": 8100,
     "@OutputResolution": "720P",
     "@SamplingRate": 0.75 }
  },
  {"id": "1",
   "input_transform_id": "0",
   "class_name": "OpenALPR",
   "parameters": {
     "@MinSize": 100,
     "@MaxSize": 1000,
     "@Step": 10 }
  } ]
```

Figure 2: VideoStorm **Query for license plate reader.**

see Figure 1. Every query is a DAG of *transforms* on *live video* that is continuously streamed to the cluster; each transform processes a time-ordered stream of messages (e.g., video frames) and passes its outputs downstream.

Figure 1 shows two example queries. One query runs across two machines; after decoding the video and subtracting the background, it sends the detected objects to another machine for tracking and classification. The other query for detecting license plates runs on a single machine. We assume there is sufficient bandwidth provisioned for cameras to stream their videos into the cluster.

Every worker machine runs a *machine manager* which start *worker processes* to host transforms. The machine manager periodically reports resource utilizations as well as status of the running transforms to the VideoStorm manager. The scheduler in the manager uses this information to allocate resources to queries. The VideoStorm manager and the machine managers are not on the query *data path*; videos are streamed directly to the decoding transforms and thereon between the transforms.

2.2 Video Queries Specification

Queries submitted to the VideoStorm manager are strung together as pipelines of *transforms*. Figure 2 shows a sample VideoStorm pipeline with two transforms. The first transform decodes the live video to produce frames that are pushed to the second transform to find license plate numbers using the OpenALPR library [13].

Each transform contains an id and class_name which is the class implementing the transform (§7). The input_transform_id field specifies the transform whose output feeds into this transform, thus allowing us to describe a pipeline. VideoStorm allows arbitrary DAGs including multiple inputs and outputs for a transform. *Source* transforms, such as the "Decoder", do not specify input transform, but instead directly connect to the camera source (specified using IP and port number).

C	D	Q
A1	1	0.6
A2	2	0.7
A3	3	0.8

(a) Query A

C	D	Q
B1	1	0.1
B2	2	0.3
B3	3	0.9

(b) Query B

		Query A					Query B				
Time	R	C	D	A	Q	L	C	D	A	Q	L
0	4	A2	2	2	0.7	-	B2	2	2	0.3	-
10	2	A1	1	1	0.6	-	B1	1	1	0.1	-
22	4	A2	2	2	0.7	-	B2	2	2	0.3	-

(c) Fair allocation

		Query A					Query B				
Time	R	C	D	A	Q	L	C	D	A	Q	L
0	4	A1	1	1	0.6	-	B3	3	3	0.9	-
10	2	A1	1	1	0.6	-	B3	3	1	0.9	-
22	4	A1	1	1	0.6	-	B2	2	3	0.3	8s
38	4	A1	1	1	0.6	-	B3	3	3	0.9	-

(d) Performance-based allocation

Table 1: Tables (a) and (b) show queries A and B with three configurations each, resource demand *D* and quality *Q*. Tables (c) and (d) show the time and capacity *R*, and for each query the chosen configuration *C*, demand *D*, allocation *A*, achieved quality *Q*, and lag *L* for the fair and performance-based schedulers. Notice in (d) that query B achieves higher quality between times 10 and 22 than with the fair scheduler in (c), and never lags beyond its permissible 8s.

Each transform contains optional knobs (parameters); e.g., the minimum and maximum window sizes (in pixels) of license plates to look for and the step increments to search between these sizes for the OpenALPR transform (more in §5). Knobs whose values can updated dynamically start with the '@' symbol. The VideoStorm manager updates them as part of its scheduling decisions.

3 Making the Case for Resource Allocation

We make the case for resource management in video analytics clusters using a simple example (§3.1) and real-world video queries (§3.2).

3.1 Motivating Example

Cluster managers such as Yarn [3], Apollo [31] and Mesos [49] commonly divide resources among multiple queries based on *resource fairness*. Being agnostic to query quality and lag preferences, fair allocation is the best they can do. Instead, *scheduling for performance*

leads to queries achieving better quality and lag.

The desirable properties of a scheduler for video analytics are: (1) allocate more resources to queries whose qualities will improve more, (2) allow queries with built-up lag in their processing to "catch up," and (3) adjust query configuration based on the resource allocated.

Tables 1a and 1b shows two example queries A and B with three knob configurations each (Ax and Bx, respectively). Query A's improvement in quality Q is less pronounced than B's for the same increase in resource demand D. Note that D is the resource to keep up with the incoming data rate. Query A cannot tolerate any lag, but B can tolerate up to 8 seconds of lag. Lag is defined as the difference between the time of the last-arrived frame and the time of the last-processed frame, i.e., how much time's worth of frames are queued-up unprocessed.

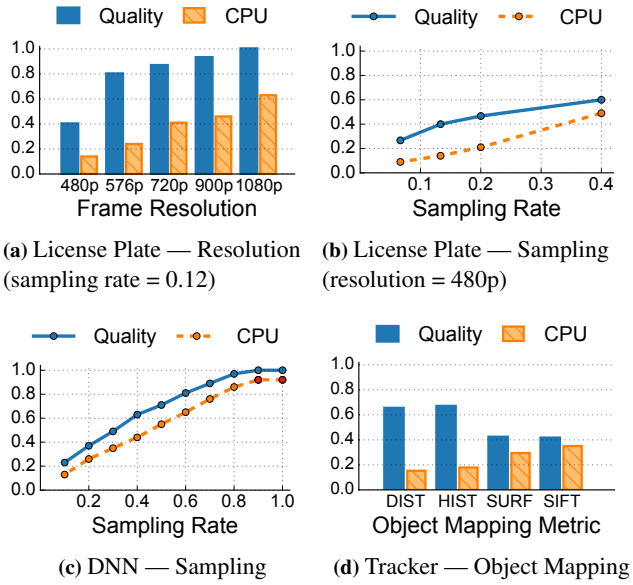
Let a single machine with resource capacity R of 4 run these two queries. Its capacity R drops to 2 after 10 seconds and then returns back to 4 after 12 more seconds (at 22 seconds). This drop could be caused by another high-priority job running on this machine.

Fair Scheduling. Table 1c shows the assigned configuration C, query demand D, resource allocation A, quality Q and lag L with a fair resource allocation. Each query selects the best configuration to keep up with the live stream (i.e., keeps its demand below its allocation). Using the fair scheduler, both queries get an allocation of 2 initially, picking configurations A2 and B2 respectively. Between times 10 to 22, when the capacity drops to 2, the queries get an allocation of 1 each, and pick configurations A1 and B1. At no point do they incur any lag.

Performance-based Scheduling. As Table 1d shows, a performance-based scheduler allocates resources of 1 and 3 to queries A and B at time 0; B can thus run at configuration B3, achieving higher quality compared to the fair allocation (while A's quality drops only by 0.1). This is because the scheduler realizes the value in providing more resources to B given its resource-quality profile.

At time 10 when capacity drops to 2, the scheduler allocates 1 unit of resource to each to the queries, but retains configuration B3 for B. Since resource demand of B3 is 3, but B has been allocated only 1, B starts to lag. Specifically, every second, the lag in processing will *increase* by $2/3$ of a second. However, query B will still produce results at quality 0.9, albeit delayed. At time 22, the capacity recovers and query B has built up a lag of 8 seconds. The scheduler allocates 3 resource units to B but switches it to configuration B2 (whose demand is only 2). This means that query B can now *catch up* – every second it can process 1.5 seconds of video. Finally, at time 38, all the lag has been eliminated and the scheduler switches B to configuration B3 (quality 0.9).

The performance-based scheduler exhibited the three properties listed above. It allocated resources to optimize for quality and allowed queries to catch up to built-up lag, while accordingly adjusting their configurations.



(a) License Plate — Resolution (sampling rate = 0.12)

(b) License Plate — Sampling (resolution = 480p)

(c) DNN — Sampling

(d) Tracker — Object Mapping

Figure 3: Resource-quality profiles for real-world video queries. For simplicity, we plot one knob at a time.

3.2 Real-world Video Queries

Video analytics queries have many *knob configurations* that affect output quality and resource demand. We highlight the resource-quality profiles of four real-world queries—*license plate reader, car counter, DNN classifier, object tracker*—of interest to the cities we are partnering with and obtained videos from their *operational* traffic cameras (§8.1). For clarity, we plot one knob at a time and keep other knobs fixed. Quality is defined as the F1 score $\in [0, 1]$ (the harmonic mean between precision and recall [83]) with reference to a *labeled ground truth*.

License Plate Reader. The OpenALPR [13] library scans the video frame to detect potential plates and then recognizes the text on plates using optical character recognition. In general, using higher video resolution and processing each frame will detect the most license plates accurately. Reducing the resolution and processing only a subset of frames (e.g., sampling rate of 0.25) dramatically reduces resource demand, but can also reduce the quality of the output (i.e., miss or incorrectly read plates). Figures 3a and 3b plots the impact of resolution and sampling rate on quality and CPU demand.[1]

Car Counter. Resolution and sampling rate are knobs that apply to almost all video queries. A car counter monitors an "area of interest" and counts cars passing the area. In general, its results are of good quality even with low resolution and sampling rates (plots omitted).

[1] Sampling rate of 0.75 drops every fourth frame from the video.

Deep Neural Network (DNN) Classifier. Vis
cessing is employing DNNs for key tasks inclu
ject detection and classification. Figure 3c p
Caffe DNN [54] model trained with the widely-u
ageNet dataset [41] to classify objects into 1,0
gories. We see a uniform increase in the quali
classification as well as resource consumption
sampling rate. As DNN models get *compressed*
reducing their resource demand at the cost of qu
compression factor presents another knob.

Object Tracker. Finally, we have also profile
ject tracker. This query continuously models th
ground" in the video, identifies foreground ob
subtracting the background, and tracks objects *across frames* using a mapping metric. The mapping metric is a key knob (Figure 3d). Objects across frames can be mapped to each other using metrics such as distance moved (DIST), color histogram similarity (HIST), or matched over SIFT [14] and SURF [15] features.

Resource-quality profiles based on knob configurations is intrinsic to video analytics queries. These queries typically identify "events" (like license plates or car accidents), and using datasets where these events are labeled, we can empirically measure precision and recall in identifying the events for different query configurations.

In contrast to approximate SQL query processing, there are no analytical models to estimate the relationship between resource demand and quality of video queries and it *depends on the specific video feeds*. For example, reducing video resolution may not reduce OpenALPR quality if the camera is zoomed in enough. Hence queries need to be *profiled* using representative video samples.

3.3 Summary and Challenges

Designing a scheduler with the desirable properties in §3.1 for real-world video queries (§3.2) is challenging.

First, the configuration space of a query can be large and there are no analytical models to estimate the resource demand and result quality of each configuration.

Second, trading off between the lag and quality goals of queries is tricky, making it challenging to define scheduling objectives *across all queries in the cluster*.

Third, resource allocation across all queries in the cluster each with many configurations is computationally intractable, presenting scalability challenges.

4 Solution Overview

The VideoStorm scheduler is split into *offline profiling* and *online* phases (Figure 4). In the offline phase, for every query, we efficiently generate its *resource-quality profile* – a small number of configurations on the Pareto

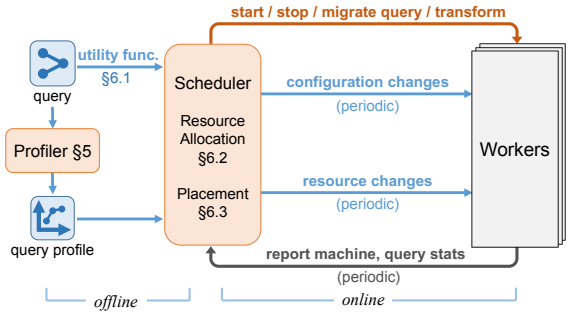


Figure 4: VideoStorm **Scheduler Components.**

curve of the profile, §5. This dramatically reduces the configurations to be considered by the scheduler.

In the online phase, the scheduler periodically (e.g., every second) considers all running queries and adjusts their resource allocation, machine placement, and configurations based on their profiles, changes in demand and/or capacity (see Figure 4). We encode the quality and lag requirements of *each individual query* into its utility function, §6.1. The performance goal across *all queries in a cluster* is specified either as maximizing the minimum utility or the sum of utilities, §6.2 and §6.3.

5 Resource-Quality Profile Estimation

When a user submits a new query, we start running it immediately with a default profile (say, from its previous runs on other cameras), while at the same time we run the query through the offline profiling phase. The query profiler has two goals. 1) Select a small subset of configurations (Pareto boundary) from the resource-quality space, and 2) Compute the query profile, $\mathcal{P}_k$, i.e., the resource demand and result quality of the selected configurations. The profile is computed either against a labeled dataset or using the initial parts of the video relative to a "golden" query configuration which might be expensive but is known to produce high-quality results.

5.1 Profile estimation is expensive

We revisit the license plate reader query from §3.2 in detail. As explained earlier, frame *resolution* and *sampling rate* are two important knobs. The query, built using the OpenALPR library [13], scans the image for license plates of size *MinSize*, then multiplicatively increases the size by *Step*, and keeps repeating this process until the size reaches *MaxSize*. The set of potential license plates is then sent to an optical character recognizer.

We estimate the quality of each knob configuration (i.e., combination of the five knobs above) on a labeled dataset using the F1 score [83], the harmonic mean between precision and recall, commonly used in machine

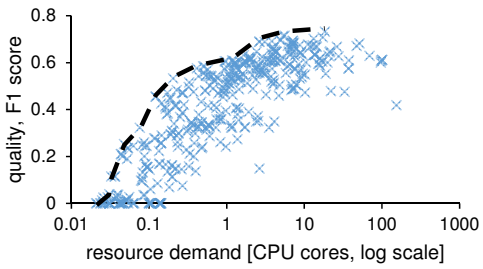


Figure 5: Resource-quality for license plate query on a 10 minute video (414 configurations); x-axis is resource demand to keep up with live video. Generating this took 20 CPU *days*. The black dashed line is the Pareto boundary.

Term	Description
$\mathcal{P}_k$	profile of query k
$c_k \in \mathcal{C}_k$	specific configuration of query k
$Q_k(c)$	quality under configuration c
$D_k(c)$	resource demand under configuration c
$L_{k,t}$	measured lag at time t
U_k	utility
Q_k^M	(min) quality goal
L_k^M	(max) lag goal
a_k	resources allocated

Table 2: Notations used, for query k.

learning; 0 and 1 represent the lowest and highest qualities. For example, increasing *MinSize* or decreasing *MaxSize* reduces the resources needed but can miss some plates and decrease quality.

Figure 5 shows a scatter plot of resource usage vs. quality of 414 configurations generated using the five knobs. There is four orders of magnitude of difference in resource usage; the most expensive configuration used all frames of a full HD resolution video and would take over 2.5 hours to analyze a 1 minute video on 1 core. Notice the vast spread in quality among configurations with similar resource usage as well as the spread in resource usage among configurations that achieve similar quality.

5.2 Greedy exploration of configurations

We implement a greedy local search to identify configuration with high quality (Q) and low demand (D); see Table 2. Our *baseline profiler* implements hill-climbing [74]; it selects a random configuration c, computes its quality $Q(c)$ and resource demand $D(c)$ by running the query with c on a *small subset* of the video dataset, and calculates $X(c) = Q(c) - \beta D(c)$ where β trades off between quality and demand. Next, we pick a neighbor configuration n (by changing the value of a *random* knob in c). If $X(n) > X(c)$, then n is better than c in quality or resource demand (or both); we set $c = n$ and repeat. When we cannot find a better neighbor (i.e., our exploration indicates that we are near a local optimum), we repeat by picking another random c.

Several enhancements significantly increase the efficiency of our search. To avoid starting with an expensive configuration and exploring its neighbors, (which are also likely to be expensive, thus wasting CPU), we pick k random configurations and start from the one with the highest $X(c)$. We found that using even $k = 3$ can successfully avoid starting in an expensive part of the search space. Second, we cache *intermediate results* in the query's DAG and reuse them in evaluating configurations with overlapping knob values.

While our simple profiler is sufficiently efficient for our purpose, sophisticated hyperparameter searches (e.g., [76]) can potentially further improve its efficiency.

Pareto boundary. We are only interested in a small subset of configurations that are on the *Pareto boundary* $\mathcal{P}$ of the resource-quality space. Let $Q(c)$ be the quality and $D(c)$ the resource demand under configuration c. If c_1 and c_2 are two configurations such that $Q(c_1) \geq Q(c_2)$ and $D(c_1) \leq D(c_2)$, then c_2 is not useful in practice; c_1 is better than c_2 in both quality and resource demand. The dashed line in Figure 5 shows the Pareto boundary of such configurations for the license plate query. We extract the Pareto boundary of the explored configurations and call it the resource-quality profile $\mathcal{P}$ of the query.

We can generate the same profile as the baseline profiler on the license plate query with 3.5× less CPU resources (i.e., 5.4 CPU hours instead of 19 CPU hours).

6 Resource Management

In the *online phase*, the VideoStorm cluster scheduler considers the utilities of individual queries and the cluster-wide performance objectives (defined in §6.1) and periodically performs two steps: resource allocation and query placement. In the resource *allocation step*, §6.2, the scheduler assumes the cluster is an aggregate bin of resources and uses an efficient heuristic to maximize the cluster-wide performance by adjusting query allocation and configuration. In the query *placement step*, §6.3, the scheduler places new queries to machines in the cluster and considers migrating existing queries.

6.1 Utility: Combining Quality and Lag

Each query has preferences on the desired quality and lag. What is the *minimum* quality goal (Q^M)? How much does the query benefit from higher quality than the goal? What is the *maximum* lag (L^M) it can tolerate and how sensitive are violations to this goal? (See Table 2 for notations.) We encode these preferences in

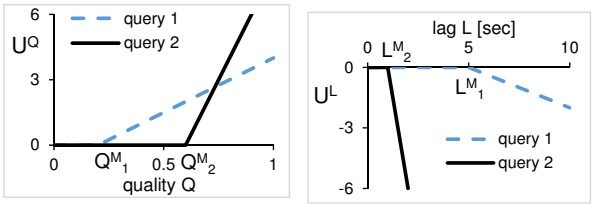


Figure 6: Examples for the second (U^Q) and third terms (U^L) in equation 1. (Left) Query 1's quality goal is relatively lenient, $Q_1^M = 0.2$, but its utility grows slowly with increase in quality beyond Q_1^M. Query 2 is more stringent, $Q_2^M = 0.6$, but its utility grows sharply thereon. (Right) Query 1 has lag target of $L_1^M = 5$ beyond which it incurs a penalty. Query 2 has a stricter lag goal of $L_2^M = 1$ and also its utility drops much faster with increased lag.

utility functions, an abstraction used extensively in economics [65, 73] and computer systems [22, 55].

Our utility function for a query has the following form, where $(x)_+$ is the positive part of x. We omit the query index k for clarity.

$$\begin{aligned} U(Q,L) &= U^B + U^Q(Q) + U^L(L) \\ &= U^B + \alpha^Q \cdot (Q - Q^M)_+ - \alpha^L \cdot (L - L^M)_+ \end{aligned} \quad (1)$$

U^B is the "baseline" utility for meeting the quality and lag goals (when $Q = Q^M$ and $L = L^M$). The second term U^Q describes how the utility responds to achieved quality Q above Q^M, the soft quality goal; the multiplier α^Q and Q^M are query-specific and set based on the application analyzing the video. Results with quality below Q^M are typically not useful to the users.

The third term, U^L, represents the penalty for results arriving later than the maximum lag goal of L^M. [2] Recall that lag is the difference between the current time and the arrival time of the last processed frame, e.g., if at time 10:30 we process a frame that arrived at 10:15, the lag is 15 minutes. Similar to latency SLOs in clusters, there is no bonus for lag being below L^M. See Figure 6 for examples of U^Q and U^L in queries.

Scheduling objectives. Given utilities of individual queries, how do we define utility or *performance* of the whole cluster? Previous work has typically aimed to maximize the minimum utility [61, 64] or sum of utilities [61, 63], which we adopt. When deployed as a "service" in the public cloud, utility will represent the revenue the cluster operator generates by executing the query; penalties and bonuses in utility translate to loss and increase in revenue. Therefore, *maximizing the sum of utilities* maximizes revenue. In a private cluster that is shared by many cooperating entities, achieving fairness is more desirable. Maximally improving the utility of the worst query provides *max-min fairness over utilities*.

[2] Multiplier α^L is in (1/second), making U^L dimensionless like U^Q.

To simplify the selection of utility functions in practical settings, we can provide only a few options to choose from. For example, the users could separately pick the minimum quality (40%, 60%, or 80%) and the maximum lag (1, 10, or 60 minutes) for a total of nine utility function *templates*. Users of cloud services already make similar decisions; for example, in Azure Storage [32], they separately select data redundancy (local, zone, or geo-distributed) and data access pattern (hot vs. cool).

6.2 Resource Allocation

Given a profile $\mathcal{P}_k$ and a utility function U_k for each query k, the scheduler allocates resources a_k to the queries and picks their query configuration ($c_k \in \mathcal{P}_k$). The scheduler runs periodically (e.g., every few seconds) and reacts to arrival of new queries, changes in query demand and lag, and changes in resource capacity (e.g., due to other high-priority non-VideoStorm jobs).

6.2.1 Scheduling Using Model-Predictive Control

The scheduler aims to maximize the minimum or sum of query utilities, which in turn depend on their quality and lag. A key point to understand is that while we can *near-instantaneously* control query quality by adjusting its configuration, query lag *accumulates* over time if we allocate less resources than query demand.

Because of this accumulation property, the scheduler cannot optimize the *current performance*, but only aims to improve performance in the *near future*. We formulate the scheduling problem using the Model-Predictive Control (MPC [67]) framework; where we model the cluster performance over a short time horizon T as a function of query configuration and allocation. In each step, we select the configuration and allocation to maximize performance over the near future (described in detail in §6.2.2).

To predict future performance, we need to predict query lag; we use the following formula:

$$L_{k,t+T}(a_k, c_k) = L_{k,t} + T - T\frac{a_k}{D_k(c_k)} \quad (2)$$

We plug in the predicted lag $L_{k,t+T}$ into the utility function (Equation 1) to obtain the predicted utility.

6.2.2 Scheduling Heuristics

We describe resource allocation assuming each query to contain only one transform, which we relax in §6.4.

Maximizing sum of utilities. The optimization problem for maximizing sum of utilities over time horizon T is as follows. Sum of allocated resources a_k cannot ex-

ceed cluster resource capacity R.

$$\max_{a_k, c_k \in \mathcal{P}_k} \quad \sum_k U_k(Q_k(c_k), L_{k,t+T}) \qquad (3)$$
$$s.t. \qquad \sum_k a_k \leq R$$

Maximizing the sum of utilities is a variant of the knapsack problem where we are trying to include the queries at different allocation and configuration to maximize the total utility. The maximization results in the best distribution of resources (as was illustrated in §3.1).

When including query k at allocation a_k and configuration c_k, we are *paying* cost of a_k and *receiving value* of $u_k = U_k(Q_k(c_k), L_{k,t+T})$. We employ a greedy approximation based on [40] where we prefer queries with highest value of u_k/a_k; i.e., we receive the largest increase in utility normalized by resource spent.

Our heuristic starts with $a_k = 0$ and in each step we consider increasing a_i (for all queries i) by a small Δ (say, 1% of a core) and consider all configurations of $c_i \in \mathcal{P}_i$. Among these options, we select query i (and corresponding c_i) with largest increase in utility.[3] We repeat this step until we run out of resources or we have selected the best configuration for each query. (Since we start with $a_k = 0$ and stop when we run out of resources, we will not end up with infeasible solutions.)

Maximizing minimum utility. Below is the optimization problem to maximize the minimum utility predicted over a short time horizon T. We require that all utilities be $\geq u$ and we maximize u.

$$\max_{a_k, c_k \in \mathcal{P}_k} \quad u \qquad (4)$$
$$s.t. \quad \forall k : U_k(Q_k(c_k), L_{k,t+T}) \geq u$$
$$\sum_k a_k \leq R$$

We can improve u only by improving the utility of the worst query. Our heuristic is thus as follows. We start with $a_k = 0$ for all queries. In each step, we select query $i = \arg\min_k U_k(Q_k(c_k), L_{k,t+T})$ with the lowest utility and increase its allocation by a small Δ, say 1% of a core. With this allocation, we compute its best configuration c_i as $\arg\max_{c \in \mathcal{P}_i} U_i(Q_i(c), L_{i,t+T})$. We repeat this process until we run out of resources or we have picked the best configuration for each query.

6.3 Query Placement

After determining resource allocation and configuration of each query, we next describe the placement of new queries and migration of existing queries. We quantify the suitability of placing a query q on machine m by computing a score for each of the following goals: high utilization, load balancing, and spreading low-lag queries.

(i) Utilization. High utilization in the cluster can be achieved by *packing* queries in to machines, thereby minimizing fragmentation and wastage of resources. Packing has several well-studied heuristics [44, 71]. We define *alignment* of a query relative to a machine using a weighted dot product, p, between the vector of machine's available resources and the query's demands; $p \in [0,1]$.

(ii) Load Balancing. Spreading load across the cluster ensures that each machine has spare capacity to handle changes in demand. We therefore prefer to place q on a machine m with the smallest utilization. We capture this in score $b = 1 - \frac{M+D}{M_{max}} \in [0,1]$, where M is the current utilization of machine m and D is demand of query q.

(iii) Lag Spreading. Not concentrating many low-lag queries on a machine provides *slack* to accumulate lag for some queries when resources are scarce, *without* having to resort to migration of queries or violation of their lag goal L^M. We achieve this by maintaining *high average* L^M on each machine. We thus compute score $l \in [0,1]$ as the average L^M *after* placing q on m.

The final score $s_{q,m}$ is the average of the three scores. For each new query q, we place it on a machine with the largest $s_{q,m}$. For each existing query q, we migrate from machine m_0 to a new machine m_1 only if its score improves substantially; i.e., $s(q,m_1) - s(q,m_0) > \tau$.

6.4 Enhancements

Incorrect resource profile. The profiled resource demand of a query, $D_k(c_k)$, might not exactly correspond to the actual query demand, e.g., when demand depends on video content. Using incorrect demand can negatively impact scheduling; for example, if $D_k(c) = 10$, but actual usage is $R_k = 100$, the scheduler would estimate that allocating $a_k = 20$ would reduce query lag at the rate of $2\times$, while the lag would actually *grow* at a rate of $5\times$. To address this, we keep track of a running average of *mis-estimation* $\mu = R_k/D_k(c)$, which represents the multiplicative error between the predicted demand and actual usage. We then incorporate μ in the lag predictor from Equation 2, $L_{k,t+T}(a_k, c_k) = L_{k,t} + T - T\frac{a_k}{D_k(c_k)}(\frac{1}{\mu})$.

Machine-level scheduling. As most queries fit on a single machine, we can respond to changes in demand or lag at the machine-level, without waiting for the cluster-wide decisions. We therefore execute the allocation step from §6.2 on each machine, which makes the scheduling logic much more scalable. The cluster-wide scheduler still runs the allocation step, but only for the purposes of determining query placement and migration.

DAG of transforms. Queries consisting of a DAG of transforms could be placed across multiple machines.

[3] We use a *concave* version of the utility functions obtained using linear interpolation to ensure that each query has a positive increase in utility, even for small Δ.

We first distribute the query resource allocation, a_k, to *individual transforms* based on per-transform resource demands. We then place individual transforms to machines as described in §6.3 while accounting for the expected data flow across machines and network link capacities.

7 VideoStorm Implementation

We now discuss VideoStorm's key implementation details and the interfaces implemented by transforms.

7.1 Implementation Details

In contrast to widely-deployed cluster frameworks like Yarn [3], Mesos [49] and Cosmos [31], we highlight the differences in VideoStorm's design. First, VideoStorm takes the list of knobs, resource-quality profiles and lag goals as inputs to allocate resources. Second, machine-level managers in the cluster frameworks *pull* work, whereas the VideoStorm manager *pushes* new queries and configuration changes to the machine-managers. Finally, VideoStorm allows machine managers to autonomously handle short-term fluctuations (§6.4)

Flow control. We implemented flow control across transforms of a query to minimize the buffering inside the query pipeline, and instead push queuing of unprocessed video to the *front* of the query. This helps for two reasons. First, decoded frames can be as much as 300× larger than the encoded video (from our benchmarks on HD videos). Buffering these frames will significantly inflate memory usage while spilling them to disk affects overall performance. Second, buffering at the front of query enables the query to respond promptly to configuration changes. It prevents frames from being processed by transforms with old inconsistent knob values.

Migration. As described in §6.3, VideoStorm migrates queries depending on the load in the cluster. We implement a simple "start-and-stop" migration where we start a copy of a running query/transform on the target machine, duplicate its input stream to the copy, and stop the old query/transform after a short period. The whole process of migration is *data-lossless* and takes roughly a second (§8.3), so the overhead of duplicated processing during the migration is very small.

Resource Enforcement. VideoStorm uses Job Objects [18] for enforcing allocations. Similar to Linux Containers [11], Job Objects allow controlling *and resizing* the CPU/memory limits of running processes.

7.2 Interfaces for Query Transforms

Transforms implement simple interfaces to process data and exchange control information.

- *Processing*. Transforms implement byte[] Process(header, data) method. header contains metadata such as frame id and timestamp. data is the input byte array, such as decoded frame. The transform returns another byte array with its result, such as the detected license plate. Each transform maintains its own state, such as the background model.
- *Configuration*. Transforms can also implement Update(key, value) to set and update knob values to change query configuration at runtime.

8 Evaluation

We evaluate the VideoStorm prototype (§7) using a cluster of 101 machines on Microsoft Azure with real video queries and video datasets. Our highlights:

1. VideoStorm outperforms the fair scheduler by 80% in quality of outputs with 7× better lag. (§8.2)
2. VideoStorm is robust to errors in query profiles and allocates nearly the same as correct profiles. (§8.3)
3. VideoStorm scales to thousands of queries with little systemic execution overheads. (§8.4)

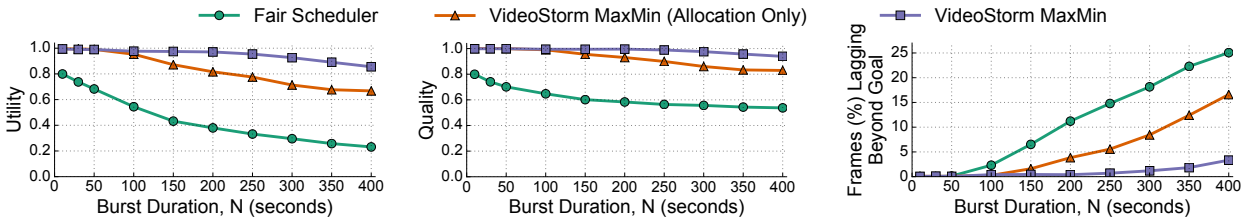
8.1 Setup

Video Analytics Queries. We evaluate VideoStorm using four types of queries described and profiled in §3.2 – license plate reader, car counter, DNN classifier, object tracker. These queries are of major interest to the cities we are partnering with in deploying our system.

Video Datasets. The above queries run on video datasets obtained from real and operational traffic cameras in Bellevue and Seattle cities for two months (Sept.–Oct., 2015). In our experiments, we stream the recorded videos at their original frame-rate (14 to 30 fps) and resolution (240P to 1080P) thereby mimicking live video streams. The videos span a variety of conditions (sunny/rainy, heavy/light traffic) that lead to variation in their processing workload. We present results on multiple different snippets from the videos.

Azure Deployment. We deploy VideoStorm on 101 D3 v2 instances on Azure's West-US cluster [6]. D3 v2 instances contain 4 cores of the 2.4GHz Intel Xeon processor and 14GB RAM. One machine ran the VideoStorm global manager on which no queries were scheduled.

Baseline. We use the work-conservative fair scheduler as our baseline. It's the widely-used scheduling policy for cluster computing frameworks like Mesos [49], Yarn [3] and Cosmos [31]. When a query, even at its best configuration, cannot use its fair share, it distributes the excess resources among the other queries. The fair scheduler places the same number of queries evenly on all available machines in a round-robin fashion.



(a) Utility **(b)** Quality **(c)** Lag

Figure 7: VideoStorm **outperforms the fair scheduler as the *duration* of burst of queries in the experiment is varied. Without its placement but only its allocation ("**VideoStorm **MaxMin (Allocation Only)"), its performance drops by a third.**

Metric. The three metrics of interest to us are quality, frames (%) exceeding the lag goal in processing, and utility (§6.1). We compare the improvement (%); if a metric (say, quality) with VideoStorm and the fair scheduler is X_V and X_f, we measure $\frac{X_V - X_f}{X_f} \times 100\%$.

8.2 Performance Improvements

Our workload consists of a mix of queries with lenient and stringent goals. We start with a set of 300 queries picked from the four types (§8.1) on 300 distinct video datasets at the beginning of the experiment. 60% of these queries have a lag goal L^M of 20s while the remaining are more lenient with a lag goal of 300s. All of them have a quality goal Q^M of 0.25. We set the lag multiplier $\alpha^L = 1$ for these long-lived video analyses.

Burst of N seconds: At a certain point, a burst of 200 license plate queries arrive and last for N seconds (which we will vary). These queries have a lag goal Q^L of 20s, a high quality goal (1.0), and higher $\alpha^L = 2$. They mimic short-term deployment of queries like AMBER Alerts with stringent accuracy and lag goals. We evaluate VideoStorm's reaction to the burst of queries up to several minutes; note that the improvements will carry over when tolerant delay and bursts are much longer.

8.2.1 Maximize the Minimum Utility (MaxMin)

We ran a series of experiments with burst duration N from 10 seconds to 400 seconds. Figure 7a plots the minimum query utility achieved in each of the experiments, when VideoStorm maximizes the minimum utility (§6.2.2). For each point in the figure, we obtain the minimum utility, quality and lag over an interval that includes a minute before and after the N second burst. VideoStorm's utility ("VideoStorm-MaxMin") drops only moderately with increasing burst duration. Its placement and resource allocations ensure it copes well with the onset of and during the burst. Contrast with the fair scheduler's sharp drop with N.

The improvement in utility comes due to smartly accounting for the resource-quality profile and lag goal of the queries; see Figures 7b and 7c. Quality (F1 score [83]; $\in [0, 1]$) with the fair scheduler is 0.2 lower than VideoStorm to begin with, but reduces significantly to nearly 0.5 for longer bursts (higher N), while quality with VideoStorm stays at 0.9, or nearly 80% better. The rest of VideoStorm's improvement comes by ensuring that despite the accumulation in lag, fewer than 5% of the frames exceed the query's lag goal whereas with the fair scheduler it grows to be 7× worse.

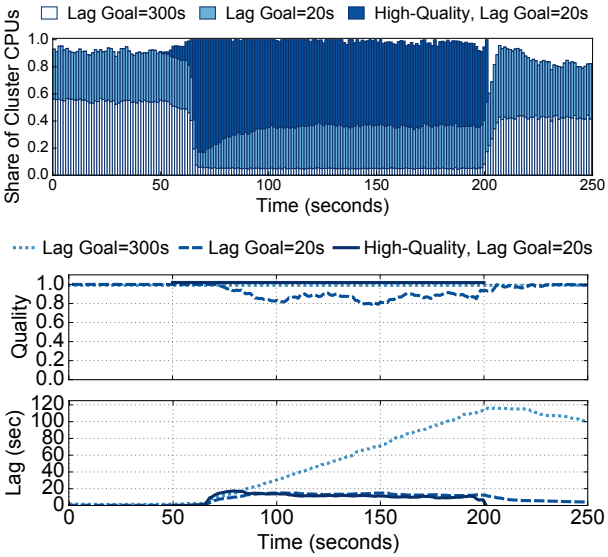


Figure 8: (Top) CPU Allocation for burst duration $N =$ 150s, and (bottom) quality and lag averaged across all queries in each of the three categories.

How valuable is VideoStorm**'s placement?** Figure 7 also shows the "VideoStorm MaxMin (Allocation Only)" graphs which lie in between the graphs for the fair scheduler and VideoStorm. As described in §6.3, VideoStorm first decides the resource allocation and then places them onto machines to achieve high utilization, load balancing and spreading of lag-sensitive and lag-tolerant queries. As the results show, not using VideoStorm's placement heuristic (instead using our baseline's round-robin placement) considerably lowers VideoStorm's gains.

Figure 8(top) explains VideoStorm's gains by plotting the allocation of CPU cores in the cluster over time, for burst duration $N = 150$s. We group the queries into

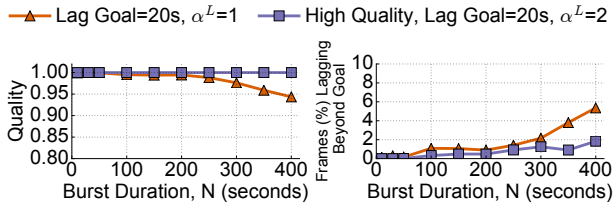
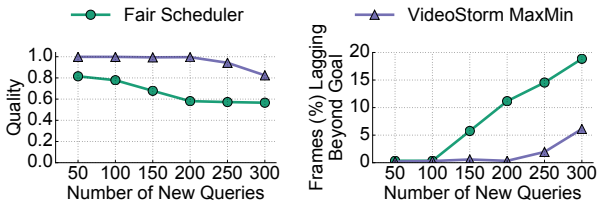


Figure 9: Impact of α^L. Queries with higher α^L have fewer frames lagging beyond their goal.



(a) Quality **(b)** Lag

Figure 10: VideoStorm **vs. fair scheduler as the number of queries in the burst during the experiment is varied.**

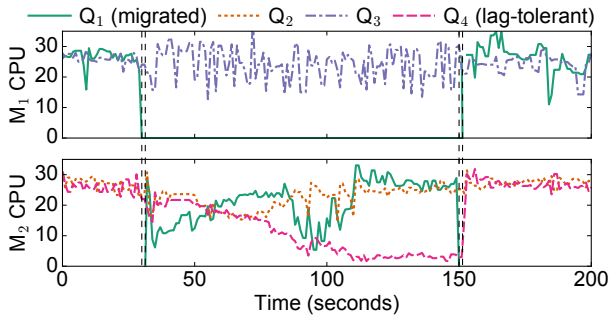


Figure 11: Q_1 migrated between M_1 and M_2. Resource for the only lag-tolerant query Q_4 (on M_2) is reduced for Q_1.

three categories — the burst of queries with 20s lag goal and quality goal of 1.0, those with 20s lag goal, and 300s lag goal (both with quality goal of 0.25). We see that VideoStorm adapts to the burst and allocates nearly 60% of the CPU cores in the cluster to the burst of license plate queries which have a high quality and tight lag goals. VideoStorm also delays processing of lag-tolerant queries (allocating less than 10% of CPUs). Figure 8(bottom) shows the resulting quality and lag, for queries in each category. We see that because the delay-tolerant queries have small allocation, their lag grows but stays below the goal. The queries with 20s lag goal reduce their quality to adapt to lower allocation and keep their lag (on average) within the bound.

Impact of α^L. Figure 9 plots the distinction in treatment of queries with the same lag goal (L^M) but different α^L and quality goals. While the figure on the left shows that VideoStorm does not drop the quality of the query with $Q^M = 1.0$, it also respects the difference in α^L; fewer frames of the query with $\alpha^L = 2$ lag beyond the goal of 20s (right). This is an example of how utility functions encode priorities.

8.2.2 Maximize the Total Utility (MaxSum)

Recall from §6.2.2 that VideoStorm can also maximize the *sum* of utilities. We measure the *average* utility, quality, and frames (%) exceeding the lag goal; maximizing for the total utility and average utility are equivalent. VideoStorm achieves 25% better quality and 5× better lag compared with the fair scheduler.

Per Query Performance. While MaxMin scheduling, as expected, results in all the queries achieving similar quality and lag, MaxSum priorities between queries as the burst duration increases. Our results show that the license plate query, whose utility over its resource demand is relatively lower, is de-prioritized with MaxSum (reduced quality as well as more frames lagging). With its high quality (1.0) and low lag (20s) goals, the scheduler has little leeway. The DNN classifier, despite having comparable resource demand does not suffer from a reduction in quality because of its tolerance to lag (300s).

8.2.3 Varying the Burst Size

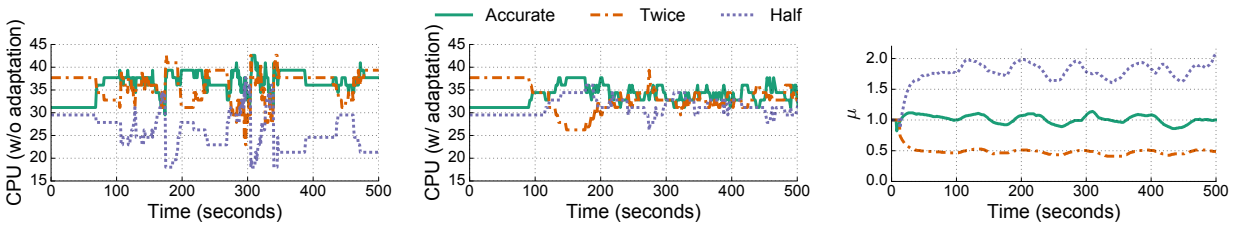
We next vary the *size* of the burst, i.e., number of queries that arrive in the burst. Note that the experiments above had varied the *duration* of the burst but with a fixed size of 200 queries. Varying the number of queries in the burst introduces different dynamics and reactions in VideoStorm's scheduler. We fix the burst duration to 200s. Figure 10 plots the results. The fair allocation causes much higher fraction of frames to exceed the lag goal when the burst size grows. VideoStorm better handles the burst and consistently performs better. Note that beyond a burst of 200 queries, resources are insufficient even to satisfy the lowest configuration (least resource demand), causing the degradation in Figure 10b.

8.3 VideoStorm's Key Features

We now highlight VideoStorm's migration of queries and accounting for errors in the resource demands.

8.3.1 Migration of Queries

Recall from §6.3 and §7 that VideoStorm migrates queries when necessary. We evaluate the value of migration by making the following addition to our experiment described at the beginning of §8.2. During the experiment, we allocate half the resources in 50% of our machines to other *non*-VideoStorm jobs. After a few minutes, the non-VideoStorm jobs complete and leave. Such jobs will be common when VideoStorm is co-situated



(a) Without Adaptation **(b)** With Adaptation **(c)** μ Over Time

Figure 12: We show three queries on a machine whose resource demands in their profiles are synthetically doubled, halved, and unchanged. By learning the proportionality factor μ (12c), our allocation adapts and converges to the right allocations (12b) as opposed to without adaptation (12a).

with other frameworks in clusters managed by Yarn [3] or Mesos [49]. We measure the migration time, and compare the performance with and without migration.

Figure 11 plots the timeline of two machines, M_1 and M_2; M_1 where a non-VideoStorm job was scheduled and M_2 being the machine *to* which a VideoStorm query Q_1, originally on M_1, was migrated. Q_1 shifts from running on M_1 to M_2 in only 1.3s. We migrate Q_1 back to M_1 when the non-VideoStorm job leaves at $\sim 150s$.

Shifting Q_1 to M_2 (and other queries whose machines were also allocated non-VideoStorm jobs, correspondingly) ensured that we did not have to degrade the quality or exceed the lag goals. Since our placement heuristic carefully spread out the queries with lenient and stringent lag goals (§6.3), we ensured that each of the machines had sufficient *slack*. As a result, when Q_1 was migrated to M_2 which already was running Q_2 and Q_4, we could delay the processing of the lag-tolerant Q_4 *without* violating any lag goals. The allocations of these delayed queries were ramped up for them to process their backlog as soon as the queries were migrated back.

As a consequence, the quality of queries with migration is 12% better than without migration. Crucially, 18× more frames (4.55% instead of 0.25%) would have exceeded the lag goal without migration.

8.3.2 Handling Errors in Query Profile

VideoStorm deals with difference between the resource demands in the resource-quality profile and the actual demand by continuously monitoring the resource consumption and adapting to errors in profiled demand (μ in §6.4). We now test the effectiveness of our correction.

We synthetically introduce *errors* in our profiles, as if they were profiles with errors, and use the erroneous profiles for our resource allocation. Consequently, the actual resource demands when the query executes do not match. In the workload above, we randomly make the profile to be half the actual resource demand for a third of the queries, twice the demand for another third, and unchanged (accurate) for the rest. VideoStorm's adaptive correction ensures that the quality and lag of queries with erroneous profiles are nearly 99.6% of results obtained *if* the profiles were perfectly accurate.

Action	Mean Duration (ms)	Standard Deviation (ms)
Start Transform	60.37	3.96
Stop Transform	3.08	0.47
Config. Change	15	2.0
Resource Change	5.7	1.5

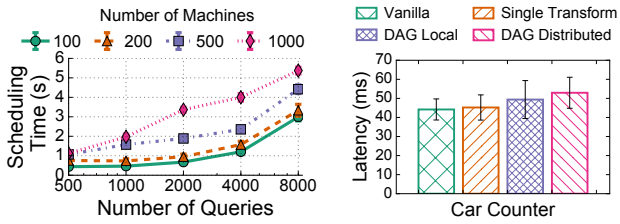
Table 3: Latency of VideoStorm's actions.

In Figure 12, we look at a single machine where VideoStorm placed three license plate queries, one each of the three different error categories. An ideal allocation (in the absence of errors) should be a third of the CPU to each of the queries. Figure 12a, however, shows how the allocation is far from converging towards the ideal *without* adaptation, because erroneous profiles undermine the precision of utility prediction. In contrast, with the adaptation, despite the errors, resource allocations converge to and stay at the ideal (Figure 12b). This is because the μ values for the queries with erroneous profiles are correctly learned as 2 and 0.5; the query without any error introduced its profile has its μ around 1 (Figure 12c).

8.4 Scalability and Efficiency

Latency of VideoStorm's actions. Table 3 shows the time taken for VideoStorm to start a new transform (shipping binaries, process startup), stop a transform, and change a 100-knob configuration and resource allocation of 10 running queries. We see that VideoStorm allows for near-instantaneous operations.

Scheduling Decisions. Figure 13a plots the time taken by VideoStorm's scheduler. Even with thousands of queries, VideoStorm make its decisions in just a few seconds. This is comparable to the scalability of schedulers in big data clusters, and video analytics clusters are unlikely to exceed them in the number of queries. Combined with the low latency of actions (Table 3), we believe VideoStorm is sufficiently scalable and agile.



(a) Scheduling Scalability **(b)** Overheads

Figure 13: Overheads in scheduling and running queries.

Transform Overheads. Finally, we measure the overhead of running a vision algorithm inside VideoStorm. We compare the latency in processing a frame while running as a vanilla process, inside a single transform, as a DAG of transforms on one machine, and as a DAG distributed across machines. Figure 13b shows that the overheads are limited. Running as a single transform, the overhead is $< 3\%$. When possible, VideoStorm places the transforms of a query DAG locally on one machine.

9 Related Work

Cluster schedulers. Cluster schedulers [3, 31, 39, 42, 44, 49, 86] do not cater to the performance objectives of streaming video analytics. They take resource demands from tasks (not the profiles), mostly allocate based on fairness/priorities, and do not resize running containers, key to dealing with resource churn in VideoStorm (§7).

Deadline-Based Scheduling. Many systems [22, 39, 42, 56, 85] adaptively allocate resources to meet deadlines of batch jobs or reduce lag of streaming queries. Scheduling in real-time systems [52, 87] has also considered using utility functions to provide (soft) deadlines to running tasks. Crucially, these systems do not consider approximation together with resource allocation to meet deadlines and do not optimize across multiple queries and servers.

Streaming and Approximate Query Processing Systems. Load shedding has been a topic of interest in streaming systems [25, 68] to manage memory usage of SQL operators but they do not consider lag in processing. Aurora, Medusa, and Borealis [19, 33, 37] and follow-up works [78, 79, 81, 82, 88] use QoS graphs to capture lag and sampling rate but they consider them *separately* and do not trade-off between them, a key aspect in our solution. In contrast to JetStream [72], that degrades data quality based on WAN bandwidths, VideoStorm identifies the best knobs to use automatically and adjusts allocations *jointly* across queries. Stream processing systems used in production [2, 4, 62, 89] do not consider load-shedding, and resource-quality tradeoff and lag in their design; Google Cloud Dataflow [21] requires manual trade-off specifications. Approximation is also used by recent [20, 23, 84] and older [47, 53] batch querying systems using statistical models for SQL operators [38].

Relative to the above literature, our main contributions are three-fold: (i) considering quality and lag of video queries *together* for multiple queries using *predictive control*, (ii) dealing with multitude of knobs in vision algorithms, and (iii) profiling *black-box vision transforms* with arbitrary user code (not standard operators).

Utility functions. Utility functions are used extensively throughout economics [65, 73], compute science [48, 55, 57, 63], and other disciplines to map how users benefit from performance [50, 58, 80]. In stream processing systems, queries describe their requirements for throughput, latency, and fraction of dropped tuples [22,34,60,79]. With multiple entities, previous work has typically maximized the minimum utility [61, 64] or sum of utilities [61, 63], which is what we also use. Utility *elicitation* [28, 30, 35] helps obtain the exact shape of the utility function.

Autonomic Computing. Autonomic computing [24, 26, 29, 66, 70, 77] allocate resources to VMs and web applications to maximize their quality of service. While some of them used look-ahead controllers based on MPC [67], they mostly ignored our main issues on the large space of configurations and quality-lag trade-offs.

10 Conclusion

VideoStorm is a video analytics system that scales to processing thousands of video streams in large clusters. Video analytics queries can adapt the quality of their results based on the resources allocated. The core aspect of VideoStorm is its scheduler that considers the resource-quality profiles of queries, each with a variety of knobs, and tolerance to lag in processing. Our scheduler optimizes jointly for the quality and lag of queries in allocating resources. VideoStorm also efficiently estimates the resource-quality profiles of queries. Deployment on an Azure cluster of 101 machines show that VideoStorm can significantly outperform a fair scheduling of resources, the widely-used policy in current clusters.

Acknowledgments

We are grateful to Xiaozhou Li, Qifan Pu, Logan Stafman and Shivaram Venkataraman for reading early versions of the draft and providing feedback. We also thank our shepherd George Porter and the anonymous NSDI reviewers for their constructive feedback. This work was partially supported by NSF Awards CNS-0953197 and IIS-1250990.

References

[1] AMBER Alert, U.S. Department of Justice. http://www.amberalert.gov/faqs.htm.

[2] Apache Flink. https://flink.apache.org/.

[3] Apache Hadoop NextGen MapReduce (YARN). https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/YARN.html.

[4] Apache Storm. https://storm.apache.org/.

[5] Avigilon. http://avigilon.com/products/.

[6] Azure Instances. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/.

[7] Capacity Scheduler. https://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[8] China's 100 Million Surveillance Cameras. https://goo.gl/UK3Obl.

[9] Genetec. https://www.genetec.com/.

[10] Hadoop Fair Scheduler. https://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[11] Linux Containers LXC Introduction. https://linuxcontainers.org/lxc/introduction/.

[12] One Surveillance Camera for Every 11 People in Britain, Says CCTV Survey. https://goo.gl/cHLqiK.

[13] Open ALPR. http://www.openalpr.com.

[14] OpenCV Documentation: Introduction to SIFT (Scale-Invariant Feature Transform). http://docs.opencv.org/3.1.0/da/df5/tutorial_py_sift_intro.html.

[15] OpenCV Documentation: Introduction to SURF (Speeded-Up Robust Features). http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_surf_intro/py_surf_intro.html.

[16] SR 520 Bridge Tolling, WA. https://www.wsdot.wa.gov/Tolling/520/default.htm.

[17] Turnpike Enterprise Toll-by-Plate, FL. https://www.tollbyplate.com/index.

[18] Windows Job Objects. https://msdn.microsoft.com/en-us/library/windows/desktop/ms684161(v=vs.85).aspx.

[19] D. J. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, Jan. 2005.

[20] S. Agarwal, B. Mozafari, A. Panda, M. H., S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *ACM EuroSys*, Apr. 2013.

[21] T. Akidau et al. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-order Data Processing. *Proceedings of the VLDB Endowment*, Aug. 2015.

[22] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive Control of Extreme-Scale Stream Processing Systems. In *IEEE ICDCS*, July 2006.

[23] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. GRASS: Trimming Stragglers in Approximation Analytics. In *USENIX NSDI*, Apr. 2014.

[24] A. AuYoung, A. Vahdat, and A. C. Snoeren. Evaluating the Impact of Inaccurate Information in Utility-Based Scheduling. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Nov. 2009.

[25] B. Babcock, M. Datar, and R. Motwani. Load Shedding for Aggregation Queries over Data Streams. In *IEEE ICDE*, Mar. 2004.

[26] A. Beloglazov and R. Buyya. Energy Efficient Resource Management in Virtualized Cloud Data Centers. In *IEEE CCGRID*, May 2010.

[27] A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica. Hierarchical Scheduling for Diverse Datacenter Workloads. In *ACM SoCC*, Nov. 2014.

[28] J. Blythe. Visual Exploration and Incremental Utility Elicitation. In *AAAI*, July 2002.

[29] N. Bobroff, A. Kochut, and K. Beaty. Dynamic Placement of Virtual Machines for Managing SLA Violations. In *IFIP/IEEE International Symposium on Integrated Network Management*, 2007.

[30] C. Boutilier, R. Patrascu, P. Poupart, and D. Schuurmans. Regret-based Utility Elicitation in Constraint-based Decision Problems. In *IJCAI*, 2005.

[31] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *USENIX OSDI*, 2014.

[32] B. Calder et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *ACM SOSP*, 2011.

[33] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams: a New Class of Data Management Applications. In *VLDB*, 2002.

[34] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator Scheduling in a Data Stream Manager. In *VLDB*, 2003.

[35] U. Chajewska, D. Koller, and R. Parr. Making Rational Decisions Using Adaptive Utility Elicitation. In *AAAI*, 2000.

[36] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. Wernsing, and D. Rob. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. In *USENIX NSDI*, 2014.

[37] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *CIDR*, Jan. 2003.

[38] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends in Databases*, Jan. 2012.

[39] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based Scheduling: If You're Late Don't Blame Us! Nov. 2014.

[40] G. B. Dantzig. Discrete-Variable Extremum Problems. *Operations Research 5 (2): 266288*, 1957.

[41] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR*, 2009.

[42] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *ACM EuroSys*, 2012.

[43] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *USENIX NSDI*, 2011.

[44] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-Resource Packing for Cluster Schedulers. In *ACM SIGCOMM*, 2014.

[45] S. Han, H. Mao, and W. J. Dally. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *Computing Research Repository*, Nov. 2015.

[46] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In *ACM MobiSys*, 2016.

[47] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *ACM SIGMOD*, 1997.

[48] A. Hernando, R. Sanz, and R. Calinescu. A Model-Based Approach to the Autonomic Management of Mobile Robot Resources. In *International Conference on Adaptive and Self-Adaptive Systems and Applications*, 2010.

[49] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *USENIX NSDI*, 2011.

[50] D. E. Irwin, L. E. Grit, and J. S. Chase. Balancing Risk and Reward in a Market-Based Task Service. In *IEEE International Symposium on High Performance Distributed Computing*, 2004.

[51] J. Jaffe. Bottleneck Flow Control. *IEEE Transactions on Communications*, 29(7):954–962, 1981.

[52] E. D. Jensen, P. Li, and B. Ravindran. On Recent Advances in Time/Utility Function Real-Time Scheduling and Resource Management. *IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, 2005.

[53] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable Approximate Query Processing with the DBO Engine. *ACM Transactions on Database Systems*, 33(4):23, 2008.

[54] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *ACM International Conference on Multimedia*, 2014.

[55] R. Johari and J. N. Tsitsiklis. Efficiency Loss in a Network Resource Allocation Game. *Mathematics of Operations Research*, 29(3):407–435, 2004.

[56] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, Í. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: towards automated SLOs for enterprise clusters. In *USENIX OSDI*, 2016.

[57] F. P. Kelly, A. K. Maulloo, and D. K. Tan. Rate Control for Communication Networks: Shadow Prices, Proportional Fairness and Stability. *Journal of the Operational Research Society*, 49(3):237–252, 1998.

[58] J. O. Kephart. Research Challenges of Autonomic Computing. In *ACM ICSE*, 2005.

[59] M. Kristan, J. Matas, A. Leonardis, M. Felsberg, L. Cehovin, G. Fernandez, T. Vojir, G. Hager, G. Nebehay, and R. Pflugfelder. The Visual Object Tracking (VOT) Challenge Results. In *IEEE ICCV Workshops*, Dec. 2015.

[60] V. Kumar, B. F. Cooper, and K. Schwan. Distributed Stream Management Using Utility-Driven Self-Adaptive Middleware. In *IEEE ICAC*, 2005.

[61] R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef. Performance management for cluster based web services. In *Integrated Network Management VIII*, pages 247–261. Springer, 2003.

[62] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. StreamScope: Continuous Reliable Distributed Processing of Big Data Streams. In *USENIX NSDI*, Mar. 2016.

[63] S. H. Low and D. E. Lapsley. Optimization Flow Control-I: Basic Algorithm and Convergence. *IEEE/ACM Transactions on Networking*, 7(6):861–874, 1999.

[64] P. Marbach. Priority Service and Max-Min Fairness. In *IEEE INFOCOM*, 2002.

[65] R. C. Merton. *Continuous-Time Finance*. Blackwell, 1990.

[66] D. Minarolli and B. Freisleben. Utility-Based Resource Allocation for Virtual Machines in Cloud Computing. In *IEEE Symposium on Computers and Communications*, pages 410–417, 2011.

[67] M. Morari and J. H. Lee. Model Predictive Control: Past, Present and Future. *Computers & Chemical Engineering*, 23(4):667–682, 1999.

[68] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *CIDR*, 2003.

[69] H. Nam and B. Han. Learning Multi-Domain Convolutional Neural Networks for Visual Tracking. *Computing Research Repository*, abs/1510.07945, 2015.

[70] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive Control of Virtualized Resources in Utility Computing Environments. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 289–302, 2007.

[71] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder. Heuristics for Vector Bin Packing. In *Microsoft Research Technical Report*, Jan. 2011.

[72] A. Rabkin, M. Arye, S. Sen, V. Pai, and M. Freedman. Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area. In *USENIX NSDI*, 2014.

[73] B. T. Ratchford. Cost-Benefit Models for Explaining Consumer Choice and Information Seeking Behavior. *Management Science*, 28, 1982.

[74] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2nd edition, 2003.

[75] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *Computing Research Repository*, abs/1409.1556, 2014.

[76] J. Snoek, H. Larochelle, and R. P. Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In *NIPS*, Dec. 2012.

[77] M. Steinder, I. Whalley, D. Carrera, I. Gaweda, and D. Chess. Server Virtualization in Autonomic Management of Heterogeneous Workloads. In *IFIP/IEEE International Symposium on Integrated Network Management*, 2007.

[78] N. Tatbul, U. Çetintemel, and S. Zdonik. Staying Fit: Efficient Load Shedding Techniques for Distributed Stream Processing. In *VLDB*, 2007.

[79] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *VLDB*, 2003.

[80] G. Tesauro, R. Das, W. E. Walsh, and J. O. Kephart. Utility-Function-Driven Resource Allocation in Autonomic Systems. In *ICAC*, 2005.

[81] Y.-C. Tu, M. Hefeeda, Y. Xia, S. Prabhakar, and S. Liu. Control-Based Quality Adaptation in Data Stream Management Systems. In *Database and Expert Systems Applications*, 2005.

[82] Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao. Load Shedding in Stream Databases: a Control-Based Approach. In *VLDB*, 2006.

[83] C. J. Van Rijsbergen. Information Retrieval. *Butterworth, 2nd edition*, 1979.

[84] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. The Power of Choice in Data-aware Cluster Scheduling. In *USENIX OSDI*, 2014.

[85] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments. In *ICAC*, 2011.

[86] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *ACM EuroSys*, 2015.

[87] E. Wandeler and L. Thiele. Real-time interfaces for interface-based design of real-time systems with fixed priority scheduling. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 80–89. ACM, 2005.

[88] Y. Wei, V. Prasad, S. H. Son, and J. A. Stankovic. Prediction-Based QoS Management for Real-Time Data Streams. In *IEEE RTSS*, 2006.

[89] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *ACM SOSP*, 2013.

Pytheas: Enabling Data-Driven Quality of Experience Optimization Using Group-Based Exploration-Exploitation

Junchen Jiang[†], *Shijie Sun*[°], *Vyas Sekar*[†], *Hui Zhang*[†⋆]
[†]*CMU,* [°]*Tsinghua University,* [⋆]*Conviva Inc.*

Abstract

Content providers are increasingly using data-driven mechanisms to optimize quality of experience (QoE). Many existing approaches formulate this process as a *prediction* problem of learning optimal decisions (e.g., server, bitrate, relay) based on observed QoE of recent sessions. While prediction-based mechanisms have shown promising QoE improvements, they are necessarily incomplete as they: (1) suffer from many known biases (e.g., incomplete visibility) and (2) cannot respond to sudden changes (e.g., load changes). Drawing a parallel from machine learning, we argue that data-driven QoE optimization should instead be cast as a *real-time exploration and exploitation (E2)* process rather than as a prediction problem. Adopting E2 in network applications, however, introduces key architectural (e.g., how to update decisions in real time with fresh data) and algorithmic (e.g., capturing complex interactions between session features vs. QoE) challenges. We present *Pytheas*, a framework which addresses these challenges using a *group-based E2* mechanism. The insight is that application sessions sharing the same features (e.g., IP prefix, location) can be grouped so that we can run E2 algorithms at a per-group granularity. This naturally captures the complex interactions and is amenable to real-time control with fresh measurements. Using an end-to-end implementation and a proof-of-concept deployment in CloudLab, we show that Pytheas improves video QoE over a state-of-the-art prediction-based system by up to 31% on average and 78% on 90th percentile of per-session QoE.

1 Introduction

We observe an increasing trend toward the adoption of data-driven approaches to optimize application-level quality of experience (QoE) (e.g., [26, 19, 36]). At a high level, these approaches use the observed QoE of recent application sessions to proactively improve the QoE for future sessions. Many previous and ongoing efforts have demonstrated the potential of such data-driven optimization for applications such as video streaming [25, 19], Internet telephony [24] and web performance [30, 39].

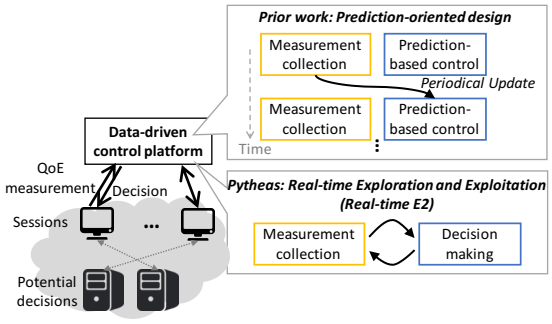


Figure 1: ***Prediction-oriented designs vs. Real-time E2.***

Existing frameworks typically formulate data-driven optimization as a *prediction* problem (e.g., [25, 39]). As depicted in Figure 1, they use passive measurements from application sessions to periodically update a prediction model, which captures the interaction between possible "decisions" (e.g., server, relay, CDN, or bitrate), different network- and application-layer features (e.g., client AS, content type), and QoE metrics. The prediction model is then used to make decisions for future sessions that yield the best (predicted) QoE.

While previous work has shown considerable QoE improvement using this prediction-based workflow, it has key limitations. First, QoE measurements are collected passively by a process independently of the prediction process. As a result, quality predictions could easily be biased if the measurement collection does not present the representative view across sessions. Furthermore, predictions could be inaccurate if quality measurements have high variance [39]. Second, the predictions and decisions are updated periodically on coarse timescales of minutes or even hours. This means that they cannot detect and adapt to sudden events such as load-induced quality degradation.

To address these concerns, we argue that data-driven optimization should instead be viewed through the lens of *real-time exploration and exploitation (E2)* frameworks [43] from the machine learning community rather than as prediction problems. At a high level (Figure 1), E2 formulates measurement collection and decision making as a *joint* process with real-time QoE mea-

surements. The intuition is to continuously strike a dynamic balance between exploring suboptimal decisions and exploiting currently optimal decisions, thus fundamentally addressing the aforementioned shortcomings of prediction-based approaches.

While E2 is arguably the right abstraction for data-driven QoE optimization, realizing it in practice in a networking context is challenging on two fronts:

- *Challenge 1: How to run real-time E2 over millions of globally distributed sessions with fresh data?* E2 techniques require a fresh and global view of QoE measurements. Unfortunately, existing solutions pose fundamental tradeoffs between data freshness and global views. While some (e.g., [19, 25]) only update the global view on timescales of minutes, others (e.g., [16]) assume that a fresh global view can be maintained by a single cluster in a centralized manner, which may not be possible for geo-distributed services [34, 33].
- *Challenge 2: How to capture complex relations between sessions in E2?* Traditional E2 techniques assume that the outcome of using some decision follows a static (but unknown) distribution, but application QoE often varies across sessions, depending on complex network- and application-specific "contexts" (e.g., last-mile link, client devices) and over time.

We observe an opportunity to address them in a networking context by leveraging the notion of **group-based E2**. This is driven by the domain-specific insight that network sessions sharing the same network- and application-level features intuitively will exhibit the same QoE behavior across different possible decisions [25, 24, 19]. This enables us to: (1) *decompose* the global E2 process into separate per-group E2 processes, which can then be independently run in geo-distributed clusters (offered by many application providers [33]); and (2) reuse well-known E2 algorithms (e.g., [14]) and run them at a coarser *per-group* granularity.

Building on this insight, we have designed and implemented *Pytheas*[1], a framework for enabling E2-based data-driven QoE optimization of networked applications. Our implementation [9] synthesizes and extends industry-standard data processing platforms (e.g., Spark [10], Kafka [2]). It provides interfaces through which application sessions can send QoE measurement to update the real-time global view and receive control decisions (e.g., CDN and bitrate selection) made by Pytheas. We demonstrate the practical benefit of Pytheas by using video streaming as a concrete use case. We extended an open source video player [5], and used a trace-driven evaluation methodology in a CloudLab-based deployment [4]. We show that (1) Pytheas improves video QoE over a state-of-the-art prediction-based baseline by 6-31% on average, and 24-78% on 90th percentile, and (2) Pytheas is horizontally scalable and resilient to various failure modes.

[1] Pytheas was an early explorer who is claimed to be the first person to record seeing the Midnight Sun.

2 Background

Drawing on prior work, we discuss several applications that have benefited (and could benefit) from data-driven QoE optimization (§2.1). Then, we highlight key limitations of existing prediction-based approaches (§2.2).

2.1 Use cases

We begin by describing several application settings to highlight the types of "decisions" and the improvements that data-driven QoE optimization can provide.

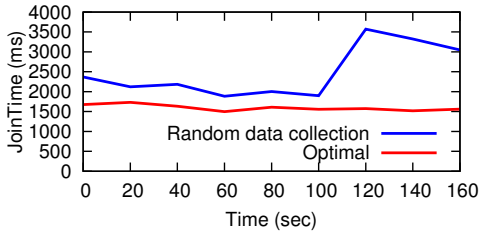
Video on demand and live streaming: VoD providers (e.g., YouTube, Vevo) have the flexibility to stream content from multiple servers or CDNs [41] and in different bitrates [23], and already have real-time visibility into video quality through client-side instrumentation [17]. Prior work has shown that compared with traditional approaches that operate on a single-session basis, data-driven approaches for bitrate and CDN selection could improve video quality (e.g., 50% less buffering time) [19, 25, 40]. Similarly, the QoE of live streaming (e.g., ESPN, twitch.tv) depends on the overlay path between source and edge servers [27]. There could be room for improvement when these overlay paths are dynamically chosen to cope with workload and quality fluctuation [32].

Internet telephony: Today, VoIP applications (e.g., Hangouts and Skype) use managed relays to optimize network performance between clients [24]. Prior work shows that compared with Anycast-based relay selection, a data-driven relay selection algorithm can reduce the number of poor-quality calls (e.g., 42% fewer calls with over 1.2% packet loss) [24, 21].

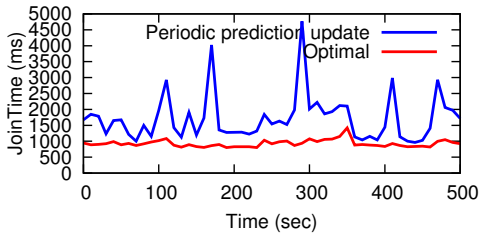
File sharing: File sharing applications (e.g., Dropbox) have the flexibility to allow each client to fetch files [18] from a chosen server or data center. By using data-driven approaches to predict the throughput between a client and a server [40, 39, 44], we could potentially improve the QoE for these applications.

Social, search, and web services: Social network applications try to optimize server selection, so that relevant information (e.g., friends' feeds) is co-located. Prior work shows that optimal caching and server selection can reduce Facebook query time by 50% [37]. Online services (e.g., search) can also benefit from data-driven techniques. For instance, data-driven edge sever selection can reduce search latency by 60% compared to Anycast, while serving 2× more queries [30].

Drawing on these use cases, we see that data-driven QoE optimization can be applied to a broad class of net-



(a) Example A: Suboptimal quality due to fixed random data collection.



(b) Example B: Overload and oscillations between decisions due to periodic prediction.

Figure 2: ***Limitations of prediction-oriented abstraction (e.g., CFA [25]) manifested in two real examples.***

worked applications that meet two requirements: (i) application providers can make decisions (e.g., server, bitrate, relay) that impact QoE of individual application sessions; and (ii) each session can measure its QoE and report it to application provider in near real time. Note that this decision-making process can either be run by the application provider itself (e.g., Skype [24]) or by some third-party services (e.g., Conviva [19]).

2.2 Limitations of predictive approaches

Many prior approaches (e.g., [19, 25, 40, 24]) for data-driven QoE optimization use a *prediction-based* workflow. That is, they periodically train a quality prediction model based on passive measurements to inform decisions for future sessions; e.g., using history data to decide what will be the best relay server for a Skype call or the best CDN for a video session? While such prediction-based approaches have proved useful, they suffer from well-known limitations, namely, *prediction bias* and *slow reaction* [22, 39]. Next, we highlight these issues using CDN selection in video streaming as a concrete use case.

2.2.1 Limitation 1: Prediction bias

A well-known problem of prediction-based workflows is that the prediction can be biased by prior decisions. Because the input measurement data are based on previous set of best decisions, we will not have a reliable way to estimate the potential quality improvements of other decisions in the future [39]. A simple solution is to use a fixed percentage of sessions to explore different decisions. This could eliminate the above prediction bias. However, it can still be suboptimal, since it might either let too many sessions use suboptimal decisions when quality is stable, or collect insufficient data in presence of higher variance.

Example A: Figure 2a shows a trace-driven evaluation to highlight such prediction biases. We use a trace of one of the major video providers in US. As a baseline, we consider prior work called CFA [25], which uses a fixed fraction of 10% sessions to randomly explore suboptimal decisions.[2] We see that it leads to worse average video startup latency, or join time, than an optimal strategy that always picks the CDN with the best average quality in each minute. Each video session can pick CDN1 or CDN2, and in the hindsight, CDN1 is on average better CDN2, except between t=40 and t=120, when CDN2 has a large variance. Even when CDN1 is consistently better than CDN2, CFA is worse than optimal, since it always assigns 10% of sessions to use CDN2. At the same time, when CDN2 becomes a better choice, CFA cannot detect this change in a timely fashion as 10% is too small a fraction to reliably estimate quality of CDN2.

2.2.2 Limitation 2: Slow reaction

Due to the time taken to aggregate sufficient data for model building, today's prediction-based systems update quality predictions periodically on coarse timescales; e.g., CFA updates models every tens of seconds [25], and VIA updates its models every several hours [24]. This means that they cannot quickly adapt to changes in operating conditions which can cause *model drifts*. First, if there are sudden quality changes (e.g., network congestion and service outage), prediction-based approaches might result in suboptimal quality due to its slow reaction. Furthermore, such model shifts might indeed be a consequence of the slow periodic predictions; e.g., the best predicted server or CDN will receive more requests and its performance may degrade as its load increases.

Example B: We consider an AS and two CDNs. For each CDN, if it receives most sessions from the AS, it will be overloaded, and the sessions served by it will have bad quality. Figure 2b shows that CFA, which always picks the CDN that has the best quality in the last minute, has worse quality than another strategy which assigns half of sessions to each CDN. This is because CFA always overloads the CDN that has the best historical performance by assigning most sessions to it, and CFA will switch decisions only *after* quality degradation occurs, leading to oscillations and suboptimal quality.

At a high level, these limitations of prediction-based approaches arise from the logical separation between measurement collection and decision making. Next, we

[2] The process begins by assigning sessions uniformly at random to all decisions in the first minute, and after that, it assigns 90% sessions to the optimal decisions based on the last minute.

discuss what the right abstraction for data-driven QoE optimization should be to avoid these limitations.

3 QoE optimization as an Exploration-Exploitation process

To avoid these aforementioned limitations of prediction-based approaches, ideally we want a framework where decisions are updated in concert with measurement collection in real time. There is indeed a well-known abstraction in the machine learning community that captures this—exploration and exploitation (E2) processes [43]. Drawing on this parallel, we argue why data-driven QoE optimization should be cast instead as a *real-time E2* process rather than a prediction-based workflow.

Background on exploration and exploitation: An intuitive way to visualize the exploration and exploitation (E2) process is through the lens of a multi-armed bandit problem [43]. Here, a gambler pulls several slot machines, each associated with an unknown reward distribution. The goal of the gambler is to optimize the mean rewards over a sequence of pulls. Thus, there is some intrinsic *exploration* phase where the gambler tries to learn these hidden reward functions, and subsequent *exploitation* phase to maximize the reward. Note that the reward functions could change over time, and thus this is a continuous process rather than a one-time shot.

QoE optimization as E2 (Figure 3): Given this framework, we can see a natural mapping between E2 and data-driven QoE optimization. Like E2, data-driven QoE optimization observes the QoE (i.e., reward) of a decision every time the decision is used (i.e., pulled) by a session. Our goal is to maximize the overall QoE after a sequence of sessions.

Casting data-driven optimization as E2 not only provides a systematic framework for data-driven QoE optimization, but also allows us to leverage well-known algorithms (e.g., [14]) from the machine learning literature. As E2 integrates the measurement collection (exploration) and decision making (exploitation) in a joint process, we can dynamically explore decisions whose quality estimation has high uncertainty, and exploit decisions that are clearly better. For instance, in Example A, an E2 process could reduce traffic for exploration when QoE is stable (before 40 second and after 120 seconds), and raise it when QoE changes (between 40 second and 120 second). By running E2 in real time with the most up-to-date measurement data, we could detect QoE drift as soon as some sessions have experienced them, and adapt to them faster than prediction-based approaches. For instance, in Example B, real-time E2 would detect load-induced QoE degradation on CDN1 as soon as its QoE is worse than CDN2, and start switching sessions to CDN2 *before* overloading CDN1.[3]

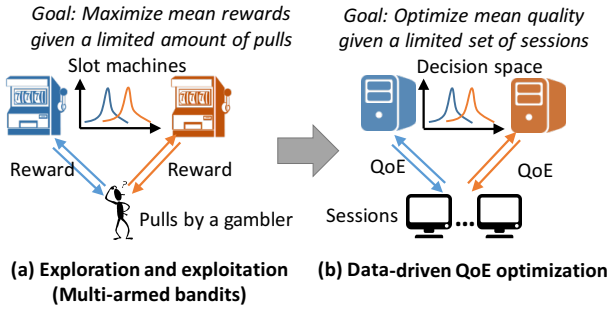


Figure 3: ***Casting data-driven QoE optimization into formulation of exploration and exploitation (E2).***

Challenges: While E2 offers the right abstraction in contrast to prediction-based approaches, applying it in network applications raises practical challenges:

- Traditional E2 techniques (e.g., [43, 29, 16]) need fresh measurements of all sessions, but getting such a fresh and global view is challenging, because application providers store fresh data in geo-distributed clusters, called *frontend clusters*, which only have a partial view across sessions. Existing analytics framework for such geo-distributed infrastructure, however, either trade global view for data freshness (e.g., [19]), or target query patterns of a traditional database (e.g., [33]), not millions of concurrent queries from geo-distributed clients, as in our case.
- Traditional E2 techniques also make strong assumptions about the context that affects the reward of a decisions, but they may not hold in network settings. For instance, they often assume some notion of continuity in context (e.g., [38]), but even when some video sessions match on all aspects of ISP, location, CDN resource availability, they may still see very different QoE, if they differ on certain key feature (e.g., last-hop connection) [25].

4 Pytheas system overview

To address the practical challenges of applying E2 in network applications, we observe a key domain-specific insight in networked applications that enables us to address both challenges in practice. We highlight the intuition behind our insight, which we refer to as group-based E2, and then provide an overview of the Pytheas system which builds on this insight.

Insight of Group-based E2: Our insight is that the "network context" of application sessions is often aligned with their "network locality". That is, if two sessions share the context that determines their E2 decisions, they will be likely to match on some network-specific fea-

[3] Note that we do not need to know the capacity of each CDN, which is often unknown to content providers.

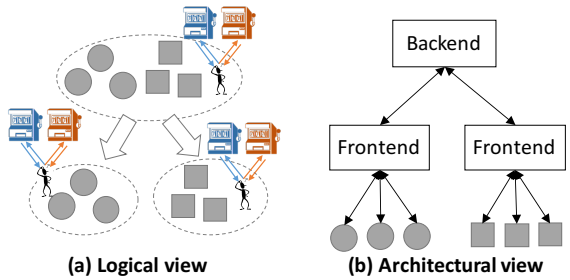


Figure 4: ***Illustration of group-based E2.***

tures. We see manifestations of this insight in many settings. For instance, video sessions with similar QoE from the same CDN/server tend to match on client IP prefix [25, 40]. Similarly, VoIP calls between the same ASes are likely to share the best relays [24], and clients from same /24 IP prefix will have similar web load time from the same edge proxy [30]. In §6.2, we validate this insight with a real-world dataset.

This insight inspires the notion of *group-based E2*, which can address the above challenges by enabling an effective decomposition of the E2 process (Figure 4a). Specifically, instead of a global E2 process over all sessions, we group together sessions with similar context by network locality and other key features (such as device and location), and use one E2 process for each group. Since sessions within a group share network locality (e.g., in the same locations and IP prefixes), they are likely to be mapped to the same frontend cluster. By running the per-group E2 logic in this frontend cluster, we can update decisions with fresh data from other sessions in the group received by this frontend cluster. Furthermore, as each group consists of sessions with similar context, it is sufficient to use traditional E2 techniques based on the data of sessions in one group, without needing a global view of all sessions. It is important to note that sessions are not grouped entirely based on IP prefixes. The sessions in the same network locality could have very different QoE, depending on the device, last-hop connectivity, and other features. Therefore, we group sessions on a finer granularity than IP prefix.

System overview: Figure 4b shows how the group-based E2 is realized in the Pytheas architecture. Each session group is managed by one per-group E2 process run by one frontend cluster. When a session comes in, it sends a request for its control decisions, which includes its features, to the Pytheas system. The request will be received by a frontend cluster, which maps the session to a group based on its features, then gets the most up-to-date decision from the local per-group E2 process, and returns the decision to the session. Each session measures its QoE and reports it to the same frontend cluster. When this frontend receives the QoE measurement, it again maps the session to a group, and updates the E2 logic of the group with the new measurement. In most cases, the E2 logic of a group is run by the same cluster that receives the requests and measurements of its sessions, so E2 logic can be updated in real time.

The backend cluster has a global, but slightly stale, view of QoE of all sessions, and it determines the session groups – which group each session belongs to and which frontend should run the per-group logic for each group. Normally, such grouping is updated periodically on a timescale of minutes. During sudden changes such as frontend cluster failures, it can also be triggered on demand to re-assign groups to frontend clusters.

The following sections will present the algorithms (§5) and system design (§6) of Pytheas, and how we implemented it (§7) in more details.

5 Pytheas algorithms

Using group-based E2, Pytheas decouples real-time E2 into two parts: a *session-grouping* logic to partition sessions into groups, and a *per-group E2* logic that makes per-session decisions. This section presents the design of these two core algorithmic pieces and how we address two issues:[4] (i) Grouping drift: the session-grouping logic should dynamically regroup sessions based on the context that determines their QoE; and (ii) QoE drift: the per-group control logic should switch decisions when QoE of some decisions change.

5.1 Session-grouping logic

Recall that sessions of the same group share the same factors on which their QoE and best decisions depend. As a concrete example, let us consider CDN selection for video. Video sessions in the same AS whose QoE depends on the local servers of different CDNs should be in the same group. However, video sessions whose QoE is bottlenecked by home wireless and thus is independent to CDNs should not be in the same group. In other words, sessions in the same group share not only the best decision, but also the factors that determine the best decisions.

A natural starting point for this grouping decision is using the notion of critical features proposed in prior work [25]. At a high level, if session A and B have the same values of critical features, they will have similar QoE. Let $S(s,F,\Delta)$ denote the set of sessions that occur within the last Δ time interval and share the same feature values as s on the set of features F, and let $Q(X)$ denote the QoE distribution of a session set X. Then, the critical feature set F^* of a session s:

$$\text{argmin}_{F\subseteq F^{all},|S(s,F,\delta)|>n}|Q(S(s,F^{all},\Delta))-Q(S(s,F,\Delta))|$$

[4] We assume in this section that the per-group control logic is updated in real time (which will be made possible in the next section).

That is, the historical session who match values on critical features F^* with s have very similar QoE distribution to those matching on all features with s on a long timescale of Δ (say last hour). The clause $|S(s,F,\delta)| > n$ ensures that there is sufficient mass in that set to get a statistically significant estimate even on small timescales δ (e.g., minutes). Such a notion of critical features has also been (implicitly) used in many other applications; e.g., AS pairs in VoIP [24] and /24 prefixes for web page load time [30]. Thus, a natural strawman for grouping algorithm is to groups sessions who match on their critical features, i.e., they have similar QoE.

However, we observe two problems inherent to critical features, which make it unsuitable to directly group sessions based on critical features: (1) First, grouping sessions based on critical features may result in groups that consist of only sessions using similar decisions, so their measurement will be biased towards a subset of decisions. (2) Second, grouping sessions based on critical features will also create overlaps between groups, so E2 logic of different groups could make conflicting decisions on these overlapping sessions. For instance, consider two Comcast sessions, s_1 and s_2, if the critical feature of s_1 is ISP, and the critical feature of s_2 is its local WiFi connection, s_2 will be in both the "WiFi" group and the "Comcast" group.

To address these issues, we formulate the goal of session grouping as following. Given a session set, the session-grouping logic should output any non-overlapping partition of sessions so that if two sessions s_1 and s_2 are in the same group, s_1 and s_2 should match values on s_1 or s_2's non-decision-specific critical features. Non-decision-specific features are the features independent of decisions; e.g., "device" is a feature independent of decisions, since video sessions of the same device can make any decisions regarding CDN and bitrate.

Operationally, we use the following approach to achieve such a grouping. First, for each session, we learn its critical features, and then ignore decision-specific features from the set of critical features of each session. Then, we recursively group sessions based on the remaining critical features in a way that avoids overlaps between groups. We start with any session s_1, and create a group consisting of all sessions that match with s_1 on s_1's critical features. We then recursively do the two following steps until every session is in some group. We find a session s_2, who is not included in any existing group, and create a new group of all sessions that match with s_2 on s_2's critical features. If the new group does not overlap with any existing group, it will be a new individual group, otherwise, we will add it to the existing groups in the way illustrated in Figure 5. We organize the existing groups in a graph, where each node is split by values of a certain feature, and each group includes

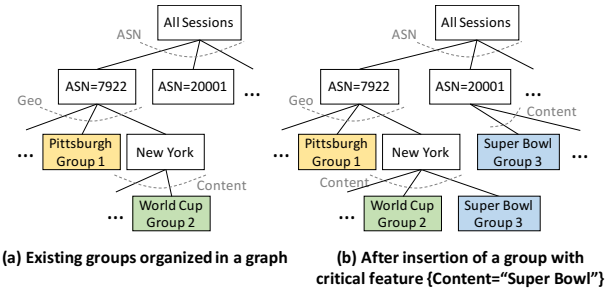


Figure 5: ***An illustrative example of session groups organized in a graph and how to a new group is added.***

multiple leaf nodes. For instance, if we want to add a new group that consists of sessions whose "content" is "Super Bowl" to a graph of existing groups as shown in Figure 5a, we will fork a path to create a new leaf node whenever the new group overlap with a existing group. Note that, this means multiple leaf nodes may be belong to the same group (e.g., "Group 3" in Figure 5b contains two different leaf nodes).

5.2 Per-group E2 logic

To run E2 in presence of QoE drift, we use Discounted UCB algorithm [20], a variant of the UCB algorithm [14], as the per-group E2 logic. UCB (Upper Confidence Bound) algorithms [14] are a family of algorithms to solve the multi-armed bandits problem. The core idea is to always opportunistically choose the arm that has the highest upper confidence bound of reward, and therefore, it will naturally tend to use arms with high expected rewards or high uncertainty. Note that the UCB algorithms do not explicitly assign sessions for "exploration" and "exploitation".

We use Discounted UCB algorithm to adapt to QoE drift, because it automatically gives more weight to more recent measurements by exponentially discounting historical measurements. Therefore, unlike other UCB algorithms which will (almost) converge to one decision, Discounted UCB is more likely to revisit suboptimal decisions to retain visibility across all decisions. We refer readers to [20] for more details. Given a session s, it returns a decision that has not been tried, if there is any. Otherwise, it calculates a score for each potential decision d by adding up an exponentially weighted moving average of d's history QoE and an estimation on the uncertainty of reward of d, and picks the decision with highest score.

6 Pytheas system architecture

Given the algorithmic pieces from the previous section, next we discuss how we map them into a system architecture. At a high level, the E2 logic of each group is independently run by frontend clusters, while the session-to-group mapping is continuously updated by the backend.

6.1 Requirements

The Pytheas system design must meet four goals:

1. *Fresh data:* The per-group E2 logic should be updated every second with newest QoE measurements.
2. *Global scale:* It should handle millions of geo-distributed sessions per second.
3. *Responsiveness:* It should respond to requests for decisions from sessions within a few milliseconds.
4. *Fault tolerance:* QoE should not be significantly impacted when parts of the system fail.

A natural starting point to achieve this goals might be to adopt a "split control plane" approach advocated by prior work for prediction-based approaches [19, 25]. At a high level, this split control plane has two parts: (1) a backend cluster that generates centralized predictions based on global but stale data, and (2) a set of geodistributed frontend servers that use these predictions from the backend to make decisions on a per-session basis. This split control architecture achieves global scale and high responsiveness, but fundamentally sacrifices data freshness.

Pytheas preserves the scale and responsiveness of the split control approach, but extends in two key ways to run group-based E2 with fresh data. First, each frontend cluster runs an active E2 algorithm rather than merely executing the (stale) prediction decisions as in prior work. Second, the frontend clusters now run per-group logic, not per-session logic. This is inspired by the insight that sessions in the same group are very likely to be received by the same frontend cluster. Thus, group-based E2 could achieve high data freshness on the session group granularity, while having the same scale and responsiveness to split control. Next, we discuss the detailed design of the frontend and backend systems.

6.2 Per-group control by frontends

The best case for group-based E2 is when all sessions of the same group are received by the same frontend cluster. When this is true, we can run the per-group E2 logic (§5.2) in real time with fresh measurements of the same group. In fact, this also is the common case. To show this, we ran session-grouping logic (§5.1) on 8.5 million video sessions in a real-world trace, and found around 200 groups each minute. Among these groups, we found that (in Figure 6) for 95% of groups, all sessions are in the same AS, and for 88% of groups, all sessions are even in the same AS *and* same city. Since existing session-to-frontend mappings (e.g., DNS or Anycast-based mechnisms) are often based on AS and geographical location, this means that for most groups, their sessions will be verly likely to be received in the same frontend clusters.

In practice, however, it is possible that sessions of one group are spread across frontend clusters. We have two

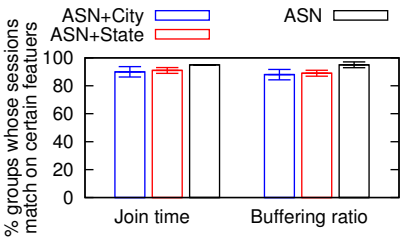


Figure 6: ***For most groups, the sessions are in the same ASN and even same city.***

options in this case:

1. Pick one cluster as the *leader* cluster of this group and let it run the E2 logic of the group based on the measurements received by this cluster. Meanwhile, other clusters, called *proxy* clusters of the group, simply receive decisions periodically from the leader cluster.
2. Keep the leader cluster and proxy clusters, but let proxy clusters not only receive decisions from the leader, but also forward QoE measurements to the leader cluster.

We see a tradeoff between the two options. While Option 1 is less complex to implement than Option 2, the leader proxy in Option 1 runs per-group logic based on only a subset of sessions, especially when the sessions of a group are evenly spread across many frontend clusters. We pick Option 2, because it is cleaner in that the per-group logic is based on all sessions in a group. In fact, implementing Option 2 does not add much complexity. Finally, Option 2 can easily fall back to Option 1 by stop forwarding measurements from proxy clusters to the leader cluster.

6.3 Updating session groups in backend

The backend cluster uses a global, stale view of measurements to update two tables, which are sent to the frontend to regroup sessions.

- First, the backend runs the session-grouping logic (§5.1) to decide which group each session belongs to, and outputs a *session-to-group table*.
- Second, it decides which frontend should be the leader cluster of each group and outputs a *group-to-leader table*. For each group, we select the frontend cluster that receives most sessions in the group as the leader.

The backend periodically (by default, every ten minutes) updates the frontend clusters with these two maps. The only exception for the maps to be updated in near real time is when one or more frontend clusters fail, which we discuss next.

6.4 Fault tolerance

As we rely on fault-tolerant components for the individual components of Pytheas within each cluster (see §7), the residual failure mode of Pytheas is when some clusters are not available. Next, we discuss how we tackle

three potential concerns in this setting.

First, if a failed frontend is the leader cluster of a group, the states of the E2 logic of the group will be lost, and we will not be able to update decisions for sessions of the group. To detect frontend failures and minimize their impact, each frontend sends frequent heartbeat messages through a "fast channel" every few seconds (by default, every five seconds) to the backend, so backend can detect frontend failures based on these heartbeat messages. Once the backend detects a frontend failure, it will select a new leader clusters for any group whose leader cluster has been the failed one, and recover the per-group logic in the new leader cluster. To recover the per-group states, each leader always shares the per-group states with its proxy clusters in the decision update messages, so that when a proxy cluster is selected as the new leader, it can recover the per-group states as they are cached locally. Note that even without a leader cluster, a proxy cluster can still respond requests with the cached decisions made by the leader cluster before it fails.

Second, the sessions who are assigned to the failed frontend will not receive control decisions. To minimize this impact, Pytheas will fall back to the native control logic. Take video streaming as an example, when Pytheas is not available, the client-side video player can fall back to the control logic built into the client-side application (e.g., local bitrate adaptation) to achieve graceful QoE degradation, rather than crash [19].

Finally, if the backend cluster is not available, Pytheas will not be able to update groups. However, Pytheas does not rely on backend to makde decisions, so clients will still receive (albeit suboptimal) decisions made by Pytheas's frontend clusters.

7 Implementation and optimization

Pytheas is open source ($\approx$ 10K lines of code across Java, python, and PHP) and can be accessed at [9]. Next, we describe the APIs for applications to integrate with Pytheas, and then describe the implementation of frontend and backend, as well as optimizations we used to remove Pytheas's performance bottlenecks.

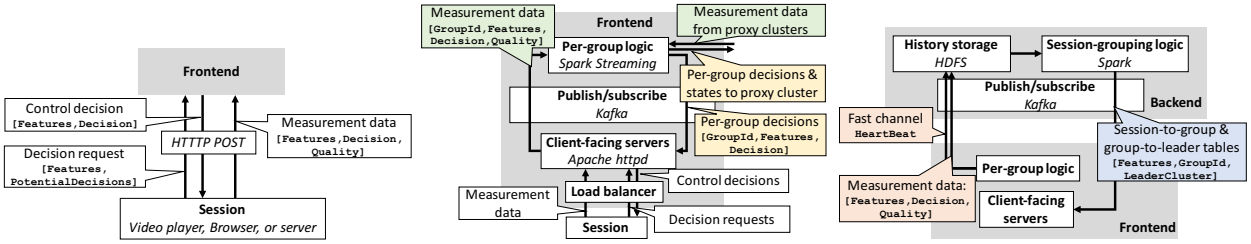
Pytheas APIs: Application sessions communicate with Pytheas through two APIs (Figure 7b): One for requesting control decisions, one for uploading measurement data. Both are implemented as standard HTTP POST messages. The session features are encoded in the data field of the POST message. Pytheas also needs content providers to provide the schema of the message that sessions send to the Pytheas frontend, as well as a list of potential decisions. Content providers may also provide QoE models that compute QoE metrics from the raw quality measurements sent by sessions.

Frontend: Figure 7b shows the key components and interfaces of a frontend cluster. When a session sends a control request to Pytheas, the request will be received by one of the client-facing servers run by Apache httpd [1]. The server processes the request with a PHP script, which first maps the session to a group and its leader cluster by matching the session features with the session-to-group and group-to-leader tables. Then the server queries the E2 logic of the group (§5.2), for the most up-to-date decision of the group, and finally returns the decision to the session. The script to process the measurement data uploaded by a session is similar; a client-facing server maps it to a group, and then forwards it to the per-group E2 logic. The per-group E2 logic is a Spark Streaming [3] program. It maintains a key-value map (a Spark RDD) between group identification and the per-group E2 states (e.g., most recent decisions), and updates the states every second by the most recent measurement data in one MapReduce operation. The communication between these processes is through Kafka [2], a distributed publish/subscribe service. We used Apache httpd, Spark Streaming, and Kafka, mainly because they are horizontally scalable and highly resilient to failures of individual machines.

While the above implementation is functionally sufficient, we observed that the frontend throughput if implemented as-is is low. Next, we discuss the optimizations to overcome the performance bottlenecks.

- *Separating logic from client-facing servers:* When a client-facing server queries the per-group control logic, the client-facing server is effectively blocked, which significantly reduces the throughput of client-facing servers. To remove this bottleneck, we add an intermediate process in each client-facing server to decouple querying control logic from responding requests. It frequently (by default every half second) pulls the fresh decision of each group from per-group logic and writes the decision in a local file of the client-facing server. Thus, the client-facing server can find the most up-to-date decisions from local cache without directly querying the control logic.
- *Replacing features with group identifier:* We found that when the number of session features increases, Spark Streaming has significantly lower throughput as it takes too long to copy the new measurement data from client-facing servers to Kafka and from Kafka to RDDs of Spark Streaming. This is avoidable, because once the features are mapped to a group by the client-facing servers, the remaining operations (updating and querying per-group logic) are completely feature-agnostic, so we can use group ID as the group identifier and remove all features from messages.

Backend: Figure 7c shows the key components and interfaces of the backend cluster. Once client-facing servers receive the measurement data from sessions, they will forward the measurement data to the backend clus-



(a) API to application sessions (b) Frontend (c) Backend

Figure 7: ***Key components and interfaces of Pytheas implementation.***

ter. On receiving these measurement data, the backend stores them in an HDFS for history data, and periodically (by default every 10 minutes) runs the session-grouping logic (§5.1) as a Spark [10] job to learn the session-group mapping and group-cluster mapping from the stored history data. These tables are sent to each frontend through Kafka, so that the future messages (requests and measurement data) from sessions will be matched against new tables. In addition, to detect failures of frontend clusters, each frontend cluster sends a small heartbeat message to the backend cluster every 5 seconds.

8 Evaluation

To evaluate Pytheas, we run our prototype [9] across multiple instances in CloudLab [4]. Each instance is a physical machine that has 8 cores (2.4 GHz) and 64GB RAM. These instances are grouped to form two frontend clusters and one backend cluster (each includes 5 to 35 instances). This testbed is an end-to-end implementation of Pytheas described in §7[5].

By running trace-driven evaluation and microbenchmarks on this testbed deployment, we show that:

- In the use case of video streaming, Pytheas improves the mean QoE by up to 6-31% and the 90th percentile QoE by 24-78%, compared to a prediction-based baseline (§8.1).
- Pytheas is horizontally scalable and has similar low response delay to existing prediction-based systems (§8.2).
- Pytheas can tolerate failures on frontend clusters by letting clients fall back to local logic and rapidly recovering lost states from other frontends (§8.3).

8.1 End-to-end evaluation

Methodology: To demonstrate the benefit of Pytheas on improving QoE, we use a real-world trace of 8.5 million video sessions collected from a major video streaming sites in US over a 24-hour period. Each video session can choose one of two CDNs. The sessions are replayed in the same chronological order as in the trace. We call a group of sessions a *unit* if they match values on AS, city, connection type, player type and content name.[6] We assume that when a video session is assigned to a CDN, its QoE would be the same to the QoE of a session that is randomly sampled from the same unit who use the same CDN in the same one-minute time window. For statistical confidence, we focus on the units which have at least 10 sessions on each CDN in each one-minute time windows for at least ten minutes. We acknowledge that our trace-driven evaluation has constraints similar to the related work, such as the assumption that QoE in a small time window is relatively stable in each unit (e.g., [24]) and a small decision space (e.g., [25]).

For each video session in the dataset, we run a DASH.js video player [5], which communicates with Pytheas using the API described in Figure 7a. To estimate the QoE of a video session, the video player does not stream video content from the selected CDN. Instead, it gets the QoE measurement from the dataset as described above.

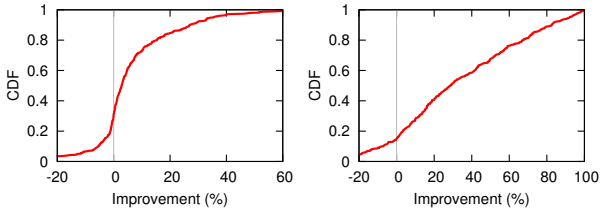
We use CFA [25] as the prediction-based baseline. It is implemented based on [25]. CFA updates QoE prediction in the backend every minute, and trains critical features every hour. The frontend clusters run a simple decision-making algorithm – for 90% sessions, it picks the decision that has the best predicted QoE, and for the rest sessions, it randomly assigns them to a CDN.

We consider two widely used video QoE metrics [17, 25]: join time (the start-up delay of a video session), and buffering ratio (the fraction of session duration spent on rebuffering). We define improvement of Pytheas for a particular unit at t-th minute by $Improve_{Pytheas}(t) = \frac{Q_{CFA}(t) - Q_{Pytheas}(t)}{Q_{CFA}(t)}$, where $Q_{Pytheas}(t)$ and $Q_{CFA}(t)$ are the average QoE of Pytheas in t-th minute and that of the baseline, respectively. Since we prefer smaller values on both metrics, a positive value means Pytheas has better QoE.

Overall improvement: Figure 8 shows the distribution of improvement of Pytheas across all sessions. We can

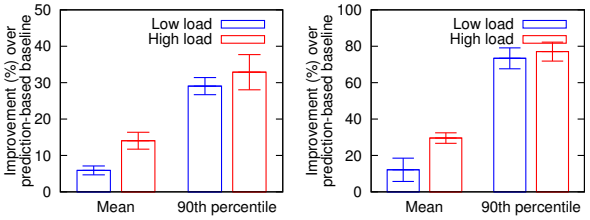
[5]Pytheas can use standard solutions such as DNS redirection to map clients to frontend clusters, and existing load balancing mechanisms provided by the host cloud service to select a frontend server instance for each client.

[6]The notion of unit is used to ensure statistical confidence of QoE evaluation, and is not used in Pytheas.



(a) Join time (b) Buffering ratio

Figure 8: ***Distribution of improvement of Pytheas over the prediction-based baseline.***



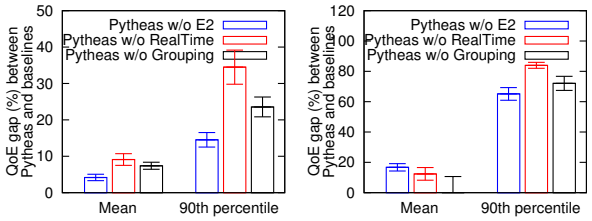
(a) Join time (b) Buffering ratio

Figure 9: ***Improvement in presence of load effect.***

see that the mean improvement is 6% for join time and 31% for buffering ratio, and the 90th percentile improvement is 24% for join time and 78% for buffering ratio. To put these numbers in context, prior studies show a 1% decrease in buffering can lead to more than a 3-minute increase in expected viewing time [17]. Note that Pytheas is not better than the baseline on every single session, because the E2 process inherently uses a (dynamic) fraction of traffic to explore suboptimal decisions.

Impact of load-induced QoE degradation: We consider the units where QoE of a CDN could significantly degrade when most sessions of the unit are assigned to use the same CDN. We assume that the QoE of a session when using a CDN under a given load (defined by the number of sessions assigned to the CDN in one minute) is the same to the QoE of a session randomly chosen in the dataset which uses the same CDN when the CDN is under a similar load. Figure 9 shows that the improvement of Pytheas when the number of sessions is large enough to overload a CDN is greater than the improvement when the number of sessions is not large enough to overload any CDN. This is because the prediction-based baseline could overload a CDN when too many sessions are assigned to the same CDN before CFA updates the QoE prediction, whereas Pytheas avoids overloading CDNs by updating its decisions in real time.

Contribution of Pytheas ideas: Having shown the overall benefit of Pytheas, we now evaluate the contribution of different components of Pytheas: (1) E2; (2) real-time update; and (3) grouping. We replace certain pieces of Pytheas by baseline solutions and compare their QoE with Pytheas's QoE. Specifically, for (1), we replace the



(a) Join time (b) Buffering ratio

Figure 10: ***Factor analysis of Pytheas ideas***

per-group E2 logic by CFA's decision-making logic; for (2), we run Pytheas with data of one-minute staleness; and for (3), we run the same E2 process over all sessions, rather than on per-group basis. Figure 10 shows the improvement of Pytheas over each baseline, and it shows that each of these ideas contributes a nontrivial improvement to Pytheas; about 10-20% improvement on average QoE and 15-80% on the 90th percentiles.

8.2 Microbenchmarks

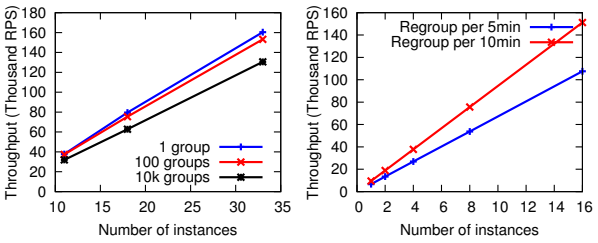
We create micro-benchmarks to evaluate the scalability and bandwidth consumption of Pytheas, as well as the benefits of various performance optimizations (§7).

8.2.1 Scalability

Frontend: Figure 11a shows the maximum number of sessions that can be served in one second, while keeping the update interval of per-group logic to be one second. Each session makes one control request and uploads QoE measurement once. Each group has the same amount of sessions. The size of control request message is 100B. We run Apache Benchmark [6] for 20 times and report the average throughput. We can see that the throughput is almost horizontally scalable to more frontend server instances. While the number of groups does impact the performance of frontend cluster, it is only to a limited extent; throughput of handling 10K groups is about 10% worse than that of handling one group.

Next, we evaluate the performance optimizations described in §7. We use a frontend cluster of 32 instances. Figure 12 shows that by separating E2 logic from client-facing servers, we can achieve 8x higher throughput, because each request reads cached decisions, which are still frequently updated. By replacing features by group identifiers, we can further increase throughput by 120%, because we can copy less data from client-facing servers to the servers running E2 logic. Note these results do not merely show the scalability of Spark Streaming or web servers; they show that our implementation of Pytheas introduces minimal additional cost, and can fully utilize existing data analytics platforms.

Backend: Figure 11b shows the maximum number of sessions that can be served in each second by a backend cluster, while keeping the completion time of session-



(a) Frontend (b) Backend

Figure 11: ***Pytheas throughput is horizontally scalable.***

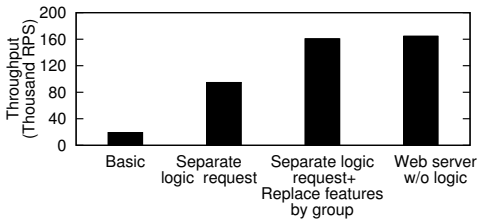


Figure 12: ***Optimizations of frontend throughput.***

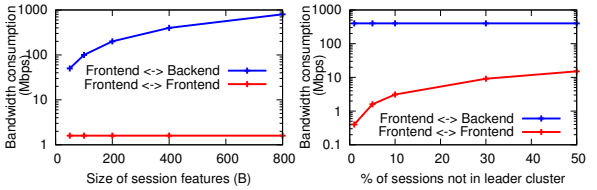
grouping logic within 5 minutes or 10 minutes. We see that the throughput is also horizontally scalable with more instances in the backend cluster.

To put the scalability numbers of both frontend and backend in context, let us consider a content provider like YouTube which has 5 billion sessions per day [11] (i.e., 57K sessions per second). Pytheas can achieve this throughput using one frontend cluster of 18 instances and a backend cluster of 8 instances, which is a tiny portion compared to the sheer number of video servers (at least on the magnitude of hundreds of thousands [7]). This might make Spark Streaming and Kafka an overkill for Pytheas, but the scale of data rate can easily increase by one to two magnitudes in real world, e.g., tens of GB/s; for instance, each video session can request tens of mid-stream decisions during an hour-long video, instead of an initial request.

8.2.2 Bandwidth consumption

Since the inter-cluster bandwidth could be costly [42, 33], we now evaluate the inter-cluster bandwidth consumption of Pytheas. We consider one session group that has one proxy cluster and one leader cluster.

First, we evaluate the impact of message size. We set the fraction of sessions received by the proxy cluster to be 5% of the group, and increase the request message size by adding more features. Figure 13a shows that the bandwidth consumption between the frontend clusters does not grow with larger message size, because the session features are replaced by group identifiers by the client-facing servers. Only the bandwidth consumption between frontend and backend grows proportionally with the message size but such overhead is inherent in existing data collection systems and is not caused by Pytheas.



(a) Impact of message size (b) Impact of % sessions not in the leader cluster

Figure 13: ***Bandwidth consumption between clusters.***

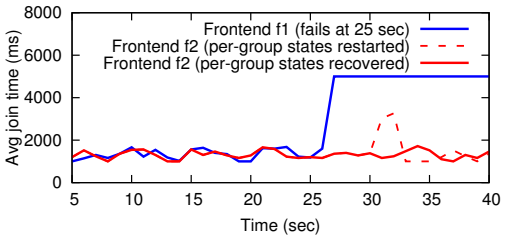


Figure 14: ***Pytheas can tolerate loss of a frontend cluster by falling back to player native logic gracefully, and recovering the logic states in a new cluster.***

Next, we evaluate the impact of fraction of sessions received by the proxy cluster. We set the message size to be 400B, and change the fraction of sessions received by each proxy cluster. Figure 13a shows that the bandwidth consumption between frontend clusters raises as more measurement data need to be forwarded from proxy to the leader cluster, but it is much smaller than the bandwidth consumption between frontend and backend.

8.3 Fault tolerance

Finally, we stress test the prototype under the condition that a leader frontend cluster fails. We set up 10 video players, each of which can stream content from two CDNs. CDN1 has 5000ms join time and CDN2 has 1000ms join time. By default, the player's native logic chooses CDN1. There are two frontend clusters, f1 and f2. The experiment begins with f1 being the leader cluster, and it loses connection at $t = 25$.

Figure 14 shows the time-series of QoE of sessions that are mapped to each frontend clusters. First, we see that the sessions mapped to f1 can fall back to the CDN chosen by the player's native logic, rather than crashing. Second, right after f1 fails, f2 should still be able to give cached decision (made by f1 before it fails) to its sessions. At $t = 30$, the backend selects f2 as the new leader for the group. At the point, a naive way to restart per-group logic in the new leader is to start it from scratch, but this will lead to suboptimal QoE at the beginning (the dotted line between $t = 30$ and $t = 35$). Pytheas avoids this cold-start problem by keeping a copy of the per-group states in the proxy cluster. This allows the proxy cluster to recover the per-group control states

without QoE degradation.

9 Related work

Data-driven QoE optimization: There is a large literature on using data-driven techniques to optimize QoE for a variety of applications, such as video streaming (e.g., [19, 25]), web service (e.g., [30, 39]), Internet telephony [21, 24], cloud services (e.g., [28]), and resource allocation (e.g., [15]). Some recent work also shows the possibility of using measurement traces to extrapolate the outcome of new system configurations [12]. Unlike these prediction-based approaches, we formulate QoE optimization as an real-time E2 process, and show that by avoiding measurement bias and enabling real-time updates, this new formulation achieves better QoE than prediction-based approaches.

Related machine learning techniques: E2 is closely related to reinforcement learning [43], where most techniques, including the per-group E2 logic used in Pytheas, are variants of the UCB1 algorithm [14], though other approaches (e.g., [13]) have been studied as well. Besides E2, Pytheas also shares the similar idea of clustering with linear mixed models [31], where a separate model is trained for each cluster of data points. While we borrow techniques from this rich literature [20, 35], our contribution is to shed light on the link between QoE optimization and the techniques of E2 and clustering, to highlight the practical challenges of adopting E2 in network applications, and to show group-based E2 as a practical way to solve these challenges. Though there have been prior attempts to cast data-driven optimization as multi-armed bandit processes in specific applications (e.g., [24]), they fall short of a practical system design.

Geo-distributed data analytics: Like Pytheas, recent work [33, 34, 42] also observes that for cost and legal considerations, many geo-distributed applications store client-generated data in globally distributed data centers. However, they focus on geo-distributed data analytics platforms that can handle general-purpose queries received by the centralized backend cluster. In contrast, Pytheas targets a different workload: data-driven QoE optimization uses a specific type of logic (i.e., E2), but has to handle requests from millions of geo-distributed sessions in real time.

10 Discussion

Handling flash crowds: Flash crowds happen when many sessions join at the same time and cause part of the resources (decisions) to be overloaded. While Pytheas can handle load-induced QoE fluctuations that occur in individual groups, overloads caused by flash crowds are different, in that they could affect sessions in multiple groups. Therefore, those affected sessions need to be regrouped immediately, but Pytheas does not support such real-time group learning. To handle flash crowds, Pytheas needs a mechanism to detect flash crowds and create groups for the affected sessions in real time.

Cost of switching decisions: The current design of Pytheas assumes there is little cost to switch the decision during the course of a session. While such assumption applies to today's DASH-based video streaming protocols [8], other applications (e.g., VoIP) may have significant cost when switching decisions in the middle of a session, so the control logic should not too sensitive to QoE fluctuations. Moreover, a content provider pays CDNs by 95th percentile traffic, so Pytheas must carefully take the traffic distribution into account as well. We intend to explore decision-making logic that is aware of these costs of switching decisions in the future.

11 Conclusions

With increasing demands of user QoE and diverse operating conditions, application providers are on an inevitable trajectory to adopt data-driven techniques. However, existing prediction-based approaches have key limitations that curtail the potential of data-driven optimization. Drawing on a parallel from machine learning, we argue that real-time exploration and exploitation is a better abstraction for this domain. In designing Pytheas, we addressed key practical challenges in applying real-time E2 to network applications. Our key insight is a *group-based E2* mechanism, where application sessions sharing the same features can be grouped so that we can run E2 at a coarser per-group granularity. Using an end-to-end implementation and proof-of-concept deployment of Pytheas in CloudLab, we showed that Pytheas improves video quality over state-of-the-art prediction-based system by 6-31% on mean, and 24-78% on tail QoE.

Acknowledgments

We appreciate the feedback by the anonymous NSDI reviewers, and thank Anirudh Badam for the caring and diligent shepherding of the paper. Junchen Jiang was supported in part by NSF award CNS-1345305 and a Juniper Networks Fellowship.

References

[1] Apache HTTP Server Project. `https://httpd.apache.org/`.

[2] Apache Kafka. `https://kafka.apache.org/`.

[3] Apache Spark Streaming. `http://spark.apache.org/streaming/`.

[4] CloudLab. `https://www.cloudlab.us/`.

[5] dash.js. `https://github.com/Dash-Industry-Forum/dash.js/wiki`.

[6] How a/b testing works. http://goo.gl/SMTT9N.

[7] How many servers does youtube. https://atkinsbookshelf.wordpress.com/tag/how-many-servers-does-youtube/.

[8] Overview of mpeg-dash standard. http://dashif.org/mpeg-dash/.

[9] Source code of Pytheas. https://github.com/nsdi2017-ddn/ddn.

[10] Spark. http://spark.incubator.apache.org/.

[11] Youtube statistics. http://fortunelords.com/youtube-statistics/.

[12] A. Agarwal, S. Bird, M. Cozowicz, L. Hoang, J. Langford, S. Lee, J. Li, D. Melamed, G. Oshri, O. Ribas, et al. A multiworld testing decision service. *arXiv preprint arXiv:1606.03966*, 2016.

[13] A. Agarwal, D. Hsu, S. Kale, J. Langford, L. Li, and R. Schapire. Taming the monster: A fast and simple algorithm for contextual bandits. In *Proceedings of The 31st International Conference on Machine Learning*, pages 1638–1646, 2014.

[14] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2–3), 2002.

[15] Y. Bao, X. Liu, and A. Pande. Data-guided approach for learning and improving user experience in computer networks. In *Proceedings of The 7th Asian Conference on Machine Learning*, pages 127–142, 2015.

[16] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. In *Conference on Innovative Data Systems Research (CIDR)*, 2015.

[17] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. A. Joseph, A. Ganjam, J. Zhan, and H. Zhang. Understanding the impact of video quality on user engagement. In *Proc. SIGCOMM*, 2011.

[18] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 481–494. ACM, 2012.

[19] A. Ganjam, F. Siddiqi, J. Zhan, I. Stoica, J. Jiang, V. Sekar, and H. Zhang. C3: Internet-scale control plane for video quality optimization. In *NSDI*. USENIX, 2015.

[20] A. Garivier and E. Moulines. On upper-confidence bound policies for non-stationary bandit problems. *arXiv preprint arXiv:0805.3415*, 2008.

[21] O. Haq and F. R. Dogar. Leveraging the Power of Cloud for Reliable Wide Area Communication. In *ACM Workshop on Hot Topics in Networks*, 2015.

[22] H. He and E. A. Garcia. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9):1263–1284, 2009.

[23] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson. A buffer-based approach to rate adaptation: evidence from a large video streaming service. In *ACM SIGCOMM 2014*.

[24] J. Jiang, R. Das, G. Anathanarayanan, P. Chou, V. Padmanabhan, V. Sekar, E. Dominique, M. Goliszewski, D. Kukoleca, R. Vafin, and H. Zhang. Via: Improving Internet Telephony Call Quality Using Predictive Relay Selection. In *ACM SIGCOMM*, 2016.

[25] J. Jiang, V. Sekar, H. Milner, D. Shepherd, I. Stoica, and H. Zhang. CFA: a practical prediction system for video QoE optimization. In *Proc. NSDI*, 2016.

[26] J. Jiang, V. Sekar, I. Stoica, and H. Zhang. Unleashing the potential of data-driven networking. In *Proceedings of 9th International Conference on COMmunication Systems & NETworkS (COMSNET)*, 2017.

[27] L. Kontothanassis, R. Sitaraman, J. Wein, D. Hong, R. Kleinberg, B. Mancuso, D. Shaw, and D. Stodolsky. A transport layer for live streaming in a content delivery network. *Proceedings of the IEEE*, 92(9):1408–1419, 2004.

[28] K. LaCurts, J. C. Mogul, H. Balakrishnan, and Y. Turner. Cicada: Introducing predictive guarantees for cloud networks. In *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*, 2014.

[29] L. Li, W. Chu, J. Langford, and R. E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pages 661–670. ACM, 2010.

[30] H. H. Liu, R. Viswanathan, M. Calder, A. Akella, R. Mahajan, J. Padhye, and M. Zhang. Efficiently delivering online services over integrated infrastructure. In *Proc. NSDI*, 2016.

[31] C. E. McCulloch and J. M. Neuhaus. *Generalized linear mixed models*. Wiley Online Library, 2001.

[32] M. K. Mukerjee, D. Naylor, J. Jiang, D. Han, S. Seshan, and H. Zhang. Practical, real-time centralized control for cdn-based live video delivery. In *ACM SIGCOMM*, pages 311–324. ACM, 2015.

[33] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica. Low latency geo-distributed data analytics. In *Proc. SIGCOMM*, 2015.

[34] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman. Aggregation and degradation in jetstream: Streaming analytics in the wide area. In *Proc. NSDI*, 2014.

[35] P. Rigollet and A. Zeevi. Nonparametric bandits with covariates. In *Proc. Conference on Learning Theory*, 2010.

[36] S. Seshan, M. Stemm, and R. H. Katz. Spand: Shared passive network performance discovery. In *USENIX Symposium on Internet Technologies and Systems*, pages 1–13, 1997.

[37] A. Shalita, B. Karrer, I. Kabiljo, A. Sharma, A. Presta, A. Adcock, H. Kllapi, and M. Stumm. Social hash: an assignment framework for optimizing distributed systems operations on social networks. In *Proc. NSDI*, 2016.

[38] A. Slivkins. Contextual bandits with similarity information. *The Journal of Machine Learning Research*, 15(1):2533–2568, 2014.

[39] M. Stemm, R. Katz, and S. Seshan. A network measurement architecture for adaptive applications. In *INFOCOM 2000*. IEEE.

[40] Y. Sun, X. Yin, J. Jiang, V. Sekar, F. Lin, N. Wang, T. Liu, and B. Sinopoli. CS2P: Improving Video Bitrate Selection and Adaptation with Data-Driven Throughput Prediction. In *to appear in Proc. SIGCOMM*, 2016.

[41] R. Torres, A. Finamore, J. R. Kim, M. Mellia, M. M. Munafo, and S. Rao. Dissecting video server selection strategies in the youtube cdn. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 248–257. IEEE, 2011.

[42] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 323–336, 2015.

[43] R. Weber et al. On the gittins index for multiarmed bandits. *The Annals of Applied Probability*, 2(4):1024–1033, 1992.

[44] Y. Zhang and N. Duffield. On the constancy of internet path properties. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 197–211. ACM, 2001.

Let it Flow: Resilient Asymmetric Load Balancing with Flowlet Switching

Erico Vanini* Rong Pan* Mohammad Alizadeh† Parvin Taheri* Tom Edsall*

*Cisco Systems †Massachusetts Institute of Technology

Abstract

Datacenter networks require efficient multi-path load balancing to achieve high bisection bandwidth. Despite much progress in recent years towards addressing this challenge, a load balancing design that is both simple to implement and resilient to network asymmetry has remained elusive. In this paper, we show that *flowlet switching*, an idea first proposed more than a decade ago, is a powerful technique for resilient load balancing with asymmetry. Flowlets have a remarkable *elasticity* property: their size changes automatically based on traffic conditions on their path. We use this insight to develop LetFlow, a very simple load balancing scheme that is resilient to asymmetry. LetFlow simply picks paths at random for flowlets and lets their elasticity naturally balance the traffic on different paths. Our extensive evaluation with real hardware and packet-level simulations shows that LetFlow is very effective. Despite being much simpler, it performs significantly better than other traffic oblivious schemes like WCMP and Presto in asymmetric scenarios, while achieving average flow completions time within 10-20% of CONGA in testbed experiments and 2× of CONGA in simulated topologies with large asymmetry and heavy traffic load.

1 Introduction

Datacenter networks must provide large bisection bandwidth to support the increasing traffic demands of applications such as big-data analytics, web services, and cloud storage. They achieve this by load balancing traffic over many paths in multi-rooted tree topologies such as Clos [13] and Fat-tree [1]. These designs are widely deployed; for instance, Google has reported on using Clos fabrics with more than 1 Pbps of bisection bandwidth in its datacenters [25].

The standard load balancing scheme in today's datacenters, Equal Cost MultiPath (ECMP) [16], randomly assigns flows to different paths using a hash taken over packet headers. ECMP is widely deployed due to its simplicity but suffers from well-known performance problems such as hash collisions and the inability to adapt to asymmetry in the network topology. A rich body of work [10, 2, 22, 23, 18, 3, 15, 21] has thus emerged on better load balancing designs for datacenter networks.

A defining feature of these designs is the information that they use to make decisions. At one end of the spectrum are designs that are oblivious to traffic conditions [16, 10, 9, 15] or rely only on local measurements [24, 20] at the switches. ECMP and Presto [15], which picks paths in round-robin fashion for fixed-sized chunks of data (called "flowcells"), fall in this category. At the other extreme are designs [2, 22, 23, 18, 3, 21, 29] that use knowledge of traffic conditions and congestion on different paths to make decisions. Two recent examples are CONGA [3] and HULA [21], which use feedback between the switches to gather path-wise congestion information and shift traffic to less-congested paths.

Load balancing schemes that require path congestion information, naturally, are much more complex. Current designs either use a centralized fabric controller [2, 8, 22] to optimize path choices frequently or require non-trivial mechanisms, at the end-hosts [23, 18] or switches [3, 21, 30], to implement end-to-end or hop-by-hop feedback. On the other hand, schemes that lack visibility into path congestion have a key drawback: they perform poorly in *asymmetric topologies* [3]. As we discuss in §2.1, the reason is that the optimal traffic split across asymmetric paths depends on (dynamically varying) traffic conditions; hence, traffic-oblivious schemes are fundamentally unable to make optimal decisions and can perform poorly in asymmetric topologies.

Asymmetry is common in practice for a variety of reasons, such as link failures and heterogeneity in network equipment [31, 12, 3]. Handling asymmetry gracefully, therefore, is important. This raises the question: *are there simple load balancing schemes that are resilient to asymmetry?* In this paper, we answer this question in the affirmative by developing LetFlow, a simple scheme that requires no state to make load balancing decisions, and yet it is very resilient to network asymmetry.

LetFlow is *extremely* simple: switches pick a path at random for each *flowlet*. That's it! A flowlet is a burst of packets that is separated in time from other bursts by a sufficient gap — called the "flowlet timeout". Flowlet switching [27, 20] was proposed over a decade ago as a way to split TCP flows across multiple paths without causing packet reordering. Remarkably, as we uncover in this paper, flowlet switching is also a powerful technique

for resilient load balancing.

The reason for this resilience is that flowlet sizes are *elastic* and change based on traffic conditions on different paths. On slow paths, with low per-flow bandwidth and high latency, flowlets tend to be smaller because there is a greater chance of a flowlet timeout (a large inter-packet gap for a flow). On fast paths, on the other hand, flowlets grow larger since flowlet timeouts are less likely to occur than on slow paths. This elasticity property is rooted in the fact that higher layer congestion control protocols like TCP react to traffic conditions on the flow's path, slowing down on congested paths (which leads to smaller flowlets) and speeding up on uncongested paths (which causes larger flowlets).

As a result of their elasticity, flowlets can compensate for inaccurate load balancing decisions, e.g., decisions that send an incorrect proportion of flowlets on different paths. Flowlets accomplish this by changing size in a way that naturally shifts traffic away from slow (congested) paths and towards fast (uncongested) paths. Since flowlet-based load balancing decisions need not be accurate, they do not require explicit path congestion information or feedback.

The only requirement is that the load balancing algorithm should not predetermine how traffic is split across paths. Instead, it should allow flowlets to "explore" different paths and determine the amount of traffic on each path automatically through their (elastic) sizes. Thus, unlike schemes such as Flare [27, 20], which attempts to achieve a *target* traffic split with flowlets, LetFlow simply chooses a path at random for each flowlet.

We make the following contributions:

- We show (§2) that simple load balancing approaches that pick paths in a traffic-oblivious manner for each flow [31] or packet [15] perform poorly in asymmetric topologies, and we uncover flowlet switching as a powerful technique for resilient load balancing in the presence of asymmetry.
- We design and implement LetFlow (§3), a simple randomized load balancing algorithm using flowlets. LetFlow is easy to implement in hardware and can be deployed without any changes to end-hosts or TCP. We describe a practical implementation in a major datacenter switch.
- We analyze (§4) how LetFlow balances load on asymmetric paths via detailed simulations and theoretical analysis. For a simplified traffic model, we show that flows with a lower rate are more likely to experience a flowlet timeout, and that LetFlow tends to move flows to less-congested paths where they achieve a higher rate.
- We evaluate (§5) LetFlow extensively in a small hardware testbed and large-scale simulations across a large number of scenarios with realistic traffic patterns, different topologies, and different transport protocols. We find that LetFlow is very effective. It achieves average flow completion times within 10-20% of CONGA [3] in a real testbed and 2× of CONGA in simulations under high asymmetry and traffic load, and performs significantly better than competing schemes such as WCMP [31] and an idealized variant of Presto [15].

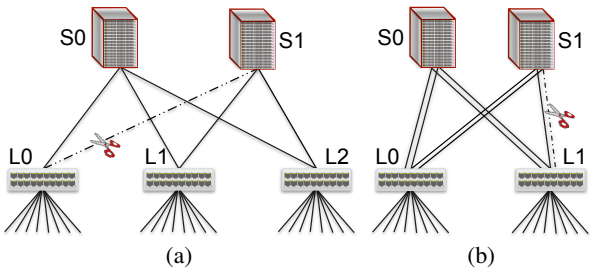


Figure 1: Two asymmetric topologies caused by link failure. All links run at 40 Gbps. Figure 1b is our baseline topology.

2 Motivation and Insights

The goal of this paper is to develop a simple load balancing scheme that is resilient to network asymmetry. In this section, we begin by describing the challenges created by asymmetry and the shortcomings of existing approaches (§2.1). We then present the key insights underlying LetFlow's flowlet-based design (§2.2).

2.1 Load balancing with Asymmetry

In asymmetric topologies, different paths between one or more source/destination pairs have different amounts of available bandwidth. Asymmetry can occur by design (e.g., in topologies with variable-length paths like BCube [14], Jellyfish [26], etc.), but most datacenter networks use symmetric topologies.[1] Nonetheless, asymmetry is difficult to avoid in practice: link failures and heterogeneous switching equipment (with different numbers of ports, link speeds, etc.) are common in large deployments and can cause asymmetry [31, 24, 3]. For instance, *imbalanced striping* [31], which occurs when the switch radix in one layer of a Clos topology is not divisible by the number of switches in an adjacent layer creates asymmetry (see [31] for details).

Figure 1 shows two basic asymmetric topologies that we will consider in this section. The asymmetry here is caused by the failure of a link. For example, in Figure 1a, the link between $L0$ and $S1$ is down, thus any $L0 \rightarrow L2$ traffic can only use the $L0 \rightarrow S0 \rightarrow L2$ path. This causes asymmetry for load balancing $L1 \rightarrow L2$ traffic.

[1] We focus on tree topologies [13, 1, 25] in this paper, since they are by far the most common in real deployments.

Scheme	Granularity	Information needed to make decisions	Handle asymmetry?
ECMP [16]	Flow	None	No
Random Packet Scatter [10]	Packet	None	No
Flare [20]	Flowlet	Local traffic	No
WCMP [31]	Flow	Topology	Partially
DRB [9]	Packet	Topology	Partially
Presto [15]	Flowcell (fixed-sized units)	Topology	Partially
LocalFlow [24]	Flow with selective splitting	Local traffic + Topology	Partially
FlowBender [18]	Flow (occasional rerouting)	Global traffic (per-flow feedback)	Yes
Hedera [2], MicroTE [8]	Flow (large flows only)	Global traffic (centralized controller)	Yes
Fastpass [22]	Packet	Global traffic (centralized arbiter)	Yes
DeTail [30]	Packet	Global traffic (hop-by-hop back-pressure)	Yes
MPTCP [23]	Packet	Global traffic (per-flow, per-path feedback)	Yes
CONGA [3]	Flowlet	Global traffic (per-path feedback)	Yes
HULA [21]	Flowlet	Global traffic (hop-by-hop probes)	Yes
LetFlow	**Flowlet**	**None (Implicit feedback via flowlet size)**	**Yes**

Table 1: Comparison of existing load balancing schemes and LetFlow. Prior designs either require explicit information about end-to-end (global) path traffic conditions, or cannot handle asymmetry.

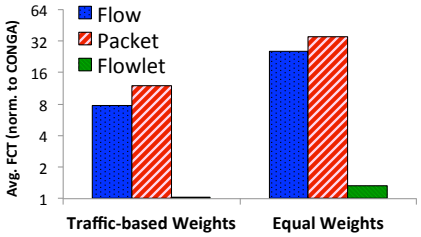


Figure 2: Load balancing dynamic traffic with asymmetry. Randomized per-flow and per-packet load balancing are significantly worse than CONGA, even with traffic-based (but static) weights; per-flowlet performs nearly as well as CONGA.

Why is asymmetry challenging? Load balancing in asymmetric topologies is difficult because the optimal split of traffic across paths in asymmetric topologies generally depends on real-time traffic demands and congestion on different paths [3]. By contrast, in symmetric topologies, splitting traffic equally across all (shortest) paths is always optimal, regardless of traffic conditions.

As an example, consider the topology in Figure 1a. Suppose the workload consists of $L0 \rightarrow L2$ and $L1 \rightarrow L2$ traffic. How should the $L1 \rightarrow L2$ traffic be split across the two paths through $S0$ and $S1$? It is not difficult to see that the ideal split depends on the amount of $L0 \rightarrow L2$ traffic. For example, if all the traffic is between $L1$ and $L2$, then we should send half of the traffic via $S0$, and half via $S1$. However, if there is 40 Gbps of $L0 \rightarrow L2$ traffic, then the $L1 \rightarrow L2$ traffic should avoid $S0$ as much as possible.

Table 1 compares several proposed load balancing schemes along two key dimensions: (1) the information they use to make load balancing decisions; and (2) the decision granularity (we discuss this aspect later). Load balancing designs that rely on *explicit* end-to-end (global) information about traffic conditions on different paths can handle asymmetry. There are many ways to collect this information with varying precision and complexity, ranging from transport-layer signals (e.g., ECN marks), centralized controllers, and in-network hop-by-hop or end-to-end feedback mechanisms. An example is CONGA [3], which uses explicit feedback loops between the top-of-rack (or "leaf") switches to collect per-path congestion information.

By contrast, schemes that are oblivious to traffic conditions generally have difficulty with asymmetry. This is the case even if different paths are weighed differently based on the topology, as some designs [31, 9, 15, 24] have proposed. Using the topology is better than nothing, but it does not address the fundamental problem of the optimal traffic splits depending on real-time traffic conditions. For instance, as we show next, knowing the topology does not help in the above scenario.

Asymmetry with dynamic workloads. Real datacenter traffic and congestion is highly dynamic [6, 7]. Since servers run at the same speed (or nearly the same speed) as network links, congestion can quickly arise as a few high rate flows start, and just as quickly dissipate as these flows end. The dynamism of real datacenter workloads makes load balancing in asymmetric topologies even more challenging, because the load balancing algorithm must adapt to fast-changing traffic conditions.

To illustrate the issue, consider again the topology in Figure 1a, and suppose that servers under switches $L0$ and $L1$ send traffic to servers under $L2$. The traffic is generated by randomly starting flows of finite length via a Poisson process, with a realistic distribution of flow sizes (details in §5). The average rate for $L0 \rightarrow L2$ and $L1 \rightarrow L2$ traffic are 20 Gbps and 48 Gbps respectively.

Ideally, the load balancer at leaf $L1$ should dynamically split traffic based on real-time $L0 \rightarrow L2$ traffic. We consider simple randomized load balancing instead, with (1) equal weights for the two paths; (2) a higher weight (roughly 2.4-to-1) for the $S1$-path compared to the $S0$-path. This weight is set statically to equalize the *average* load on the two paths. This represents a hypothetical system (e.g., a centralized controller) that optimizes the path weights using long-term traffic estimates.

Figure 2 shows the average FCT for both weight settings with load balancing decisions made on a per-flow, per-packet, and per-flowlet granularity. The results are normalized to the average FCT achieved by CONGA [3] for reference. The per-flow case is identical to ECMP and WCMP [31]. The per-packet case provides an up-

per bound on the performance of schemes like Random Packet Scatter [10] and Presto [15] that balance load on a finer granularity with static weights.[2] (We discuss the per-flowlet case later.) The results show that randomized traffic-oblivious load balancing at per-flow and per-packet granularity performs significantly worse than CONGA with both weight settings.

In summary, existing load balancing designs either explicitly use global path congestion information to make decisions or do poorly with asymmetry. Also, static weights based on the topology or coarse traffic estimates are inadequate in dynamic settings. In the next section, we describe how LetFlow overcomes these challenges.

2.2 Let the Flowlets *Flow*

Consider Figure 2 again. Remarkably, the same randomized load balancing scheme that did poorly at the per-flow and per-packet granularities performs very well when decisions are made at the level of *flowlets* [27, 20]. Picking paths uniformly at random per flowlet already does well; optimizing the path weights further improves performance and nearly matches CONGA.

Recall that flowlets are bursts of packets of the same flow that are sufficiently apart in time that they can be sent on different paths without causing packet reordering at the receiver. More precisely, the gap between two flowlets must exceed a threshold (the flowlet timeout) that is larger than the difference in latency among the paths; this ensures that packets are received in order.

Our key insight is that (in addition to enabling fine-grained load balancing) flowlets have a unique property that makes them resilient to inaccurate load balancing decisions: *flowlets automatically change size based on the extent of congestion on their path*. They shrink on slow paths (with lower per-flow bandwidth and higher latency) and expand on fast paths. This *elasticity* property allows flowlets to compensate for poor load balancing decisions by shifting traffic to less-congested paths automatically.

We demonstrate the resilience of flowlets to poor load balancing decisions with a simple experiment. Two leaf switches, $L0$-$L1$, are connected via two spine switches, $S0$-$S1$, with the path through $S1$ having half the capacity of the path through $S0$ (see topology in Figure 1b). Leaf $L0$ sends traffic — consisting of a realistic mix of short and large flows (§5) — to leaf $L1$ at an average rate of 112 Gbps (over 90% of the available 120 Gbps capacity).

We compare weighted-random load balancing on a per-flow, per-packet, and per-flowlet basis, as in the previous section. In this topology, ideally, the load balancer should pick the $S0$-path 2/3rd of the time. To model inaccurate decisions, we vary the weight (probability) of the $S0$-path from 2% to 98% in a series of simulations. At each weight, we plot the overall average flow completion time (FCT) for the three schemes, normalized to the lowest value achieved across all experiments (with per-packet load balancing using a weight of 66% for $S0$).

[2] Our implementation of per-packet load balancing includes an ideal reordering buffer at the receiver to prevent a performance penalty for TCP. See §5 for details.

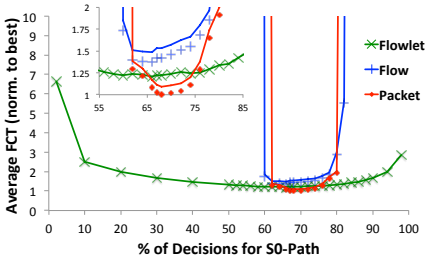


Figure 3: Flowlet-based load balancing is resilient to inaccurate load balancing decisions.

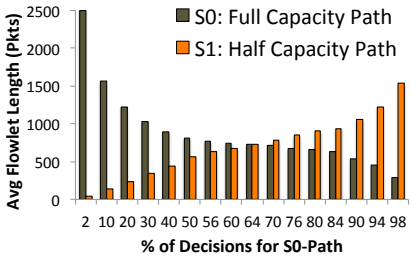


Figure 4: The flowlet sizes on the two paths change depending on how flowlets are split between them.

Figure 3 shows the results. We observe that flow- and packet-based load balancing deteriorates significantly outside a narrow range of weights near the ideal point. This is not surprising, since in these schemes, the amount of traffic on each path is directly determined by the weight chosen by the load balancer. If the traffic on either path exceeds its capacity, performance rapidly degrades.

Load balancing with flowlets, however, is robust over a wide range of weights: it is within 2× of optimal for all weights between 20–95% for $S0$. The reason is explained by Figure 4, which shows how the average flowlet size on the two paths changes based on the weights chosen by the load balancer. If the load balancer uses the correct weights (66% for $S0$, 33% for $S1$) then the flowlet sizes on the two paths are roughly the same. But if the weights are incorrect, the flowlet sizes adapt to keep the actual traffic balanced. For example, even if only 2% of flowlets are sent via $S0$, the flowlets on the $S0$ path grow $\sim 65\times$ larger than those on the $S1$ path, to keep the traffic reasonably balanced at a 57% to 43% split.

This experiment suggests that a simple scheme that spreads flowlets on different paths in a traffic-oblivious manner can be very effective. In essence, due to their elasticity, flowlets implicitly reflect path traffic conditions (we analyze precisely why flowlets are elastic in §4). LetFlow takes advantage of this property to achieve good performance without explicit information or complex feedback mechanisms.

3 Design and Hardware Implementation

LetFlow is very simple to implement. All functionality resides at the switches. The switches pick an outgoing port at random among available choices (given by the routing protocol) for each flowlet. The decisions are made on the first packet of each flowlet; subsequent packets follow the same uplink as long as the flowlet remains active (there is not a sufficiently long gap).

Flowlet detection. The switch uses a *Flowlet Table* to detect flowlets. Each entry in the table consists of a *port number*, a *valid bit* and an *age bit*. When a packet arrives, its 5-tuple header is hashed into an index for accessing an entry in the flowlet table. If the entry is active, i.e. the valid bit is set to one, the packet is sent to the port stored in the entry, and the age bit is cleared to zero. If the entry is not valid, the load balancer randomly picks a port from the available choices to forward the packet. It then sets the valid bit to one, clears the age bit, and records the chosen port in the table.

At a fixed time interval, Δ, a separate checker process examines each entry's age bit. If the age bit is not set, the checker sets the age bit to one. If the age bit is already set, the checker ages out the entry by clearing the valid bit. Any subsequent packet that is hashed to the entry will be detected as a new flowlet. This one-bit aging scheme, which was also used by CONGA [3], enables detection of flowlets without maintaining timestamps for each flow. The tradeoff is that the flowlet timeout can take any value between Δ and 2Δ. More precision can be attained by using more age bits per entry, but we find that a single bit suffices for good performance in LetFlow.

The timeout interval, Δ, plays a key role: setting it too low would risk reordering issues, but if set too high, it can reduce flowlet opportunities. We analyze the role of the flowlet timeout in LetFlow's effectiveness in §4.

Load balancing decisions. As previously discussed, LetFlow simply picks a port at random from all available ports for each new flowlet.

Hardware costs. The implementation cost (in terms of die area) is mainly for the Flowlet Table, which requires roughly 150K gates and 3 Mbits of memory, for a table with 64K entries. The area consumption is negligible ($< 0.3\%$) for a modern switching chip. LetFlow has been implemented in silicon for a major datacenter switch product line.

4 Analysis

We have seen that flowlets are resilient to inaccurate load balancing decisions because of their elasticity. In this section, we dig deeper to understand the reasons for the elasticity of flowlets (§4.1). We find that this is rooted in the fact that higher layer congestion control protocols like TCP adjust the rate of flows based on the available capacity on their path. Using these insights, we develop a Markov Chain model (§4.2) that shows how LetFlow balances load, and the role that the key parameter, the flowlet timeout, plays in LetFlow's effectiveness.

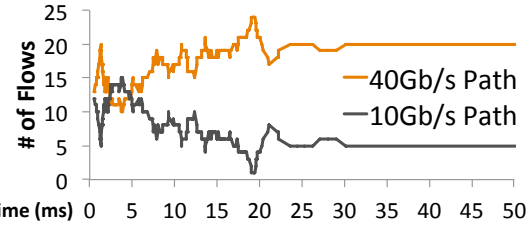


Figure 5: LetFlow balances the average rate of (long-lived) flows on asymmetric paths. In this case, 5 flows are sent to the 10 Gbps link; 20 flows to the 40 Gbps link.

4.1 Why are Flowlets elastic?

To see why flowlets are elastic, let us consider what causes flowlets in the first place. Flowlets are caused by the burstiness of TCP at sub-RTT timescale [20]. TCP tends to transmit a window of packets in one or a few clustered bursts, interspersed with idle periods during an RTT. This behavior is caused by various factors such as slow-start, ACK compression, and packet drops.

A simple explanation for why flowlet sizes change with path conditions (e.g., shrinking on congested paths) points the finger at *packet drops*. The argument goes: on congested path, more packets are dropped; each drop causes TCP to cut its window size in half, thereby idling for at least half an RTT [5] and (likely) causing a flowlet.

While this argument is valid (and easy to observe empirically), we find that flowlets are elastic for a more basic reason, rooted in the way congestion control protocols like TCP adapt to conditions on a flow's path. When a flow is sent to a slower (congested) path, the congestion control protocol reduces its rate.[3] Because the flow slows down, there is a higher chance that none of its packets arrive within the timeout period, causing the flowlet to end. On the other hand, a flow on a fast path (with higher rate) has a higher chance of having packets arrive within the timeout period, reducing flowlet opportunities.

To illustrate this point, we construct the following simulation: 25 long-lived TCP flows send traffic on two paths with bottleneck link speeds of 40 Gbps and 10 Gbps respectively. We cap the window size of the flows to 256 KB, and set the buffer size on both links to be large enough such that there are no packet drops. The flowlet timeout is set to 200 μs and the RTT is 50 μs (in absence of the queueing delay).

Figure 5 shows how the number of flows on each path

[3]This may be due to packet drops or other congestion signals like ECN marks or increased delay.

changes over time. We make two observations. First, changes to the flows' paths occur mostly in the first 30 ms; after this point, flowlet timeouts become rare and the flows reach an *equilibrium*. Second, at this equilibrium, the split of flows on the two paths is ideal: 20 flows on the 40 Gbps path, and 5 flows on 10 Gbps path. This results in an average rate of 2 Gbps per flow on each path, which is the largest possible.

The equilibrium is stable in this scenario because, in any other state, the flows on one path will have a smaller rate (on average) than the flows on the other. These flows are more likely to experience a flowlet timeout than their counterparts. Thus, the system has a natural "stochastic drift", pushing it towards the optimal equilibrium state. Notice that a stochastic drift does not guarantee that the system remains in the optimal state. In principle, a flow could change paths if a flowlet timeout occurs. But this does not happen in this scenario due to TCP's ACK-clocked transmissions. Specifically, since the TCP window sizes are fixed, ACK-clocking results in a repeating pattern of packet transmissions. Therefore, once flows reach a state where there are no flowlet timeouts for one RTT, they are locked into that state forever.

In practice, there will be far more dynamism in the cross traffic and TCP's packet transmissions and inter-packet gaps. In general, the inter-packet gap for TCP depends on the window size, RTT, and degree of intra-RTT burstiness. The precise characterization of TCP inter-packet gaps is a difficult problem. But it is clear that inter-packet gaps increase as the flow's rate decreases and the RTT increases.

In search of a simpler, non-TCP-specific model, we next analyze a simple Poisson packet arrival process that provides insight into how LetFlow balances load.

4.2 A Markov chain model

Consider n flows that transmit on two paths P_1 and P_2, with bottleneck capacity C_1 and C_2 respectively.[4] Assume packet arrivals from flow i occur as a Poisson process with rate λ_i, independent of all other flows. Poisson arrivals is not realistic for TCP traffic (which is bursty) but allows us to capture the essence of the system's dynamics while keeping the model tractable (see below for more on the limitations of the model).

Assume the flowlet timeout is given by Δ, and that each new flowlet is mapped to one of the two paths at random. Further, assume that the flows on the same path achieve an equal share of the path's capacity; i.e.,

$$\lambda_i = \begin{cases} C_1/n_1 & \text{flow } i \text{ is on path } P_1 \\ C_2/n_2 & \text{flow } i \text{ is on path } P_2 \end{cases}$$

[4]For ease of exposition, we describe the model for two paths; the general case is similar.

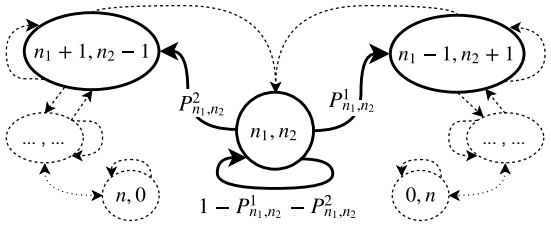


Figure 6: Markov chain model. The state $(n1, n2)$ gives the number of flows on the two paths.

where n_1 and n_2 denote the number of flows on each path. This is an idealization of the fair bandwidth allocation provided by congestion control protocols. Finally, let $\lambda_a = C_1 + C_2$ be the aggregate arrival rate on both paths.

It is not difficult to show that (n_1, n_2) forms a Markov chain. At each state, (n_1, n_2), if the next packet to arrive triggers a new flowlet for its flow, the load balancer may pick a different path for that flow (at random), causing a transition to state $(n_1 - 1, n_2 + 1)$ or $(n_1 + 1, n_2 - 1)$. If the next packet does not start a new flowlet, the flow remains on the same path and the state does not change.

Let $P^1_{n_1,n_2}$ and $P^2_{n_1,n_2}$ be the transition probabilities from (n_1, n_2) to $(n_1 - 1, n_2 + 1)$ and $(n_1 + 1, n_2 - 1)$ respectively, as shown in Figure 6. In Appendix A, we derive the following expression for the transition probabilities:

$$P^1_{n_1,n_2} = \frac{1}{2} \sum_{i \in P_1} \left[\frac{\lambda_i}{\lambda_a - \lambda_i} \left(e^{-\lambda_i \Delta} - e^{-\lambda_a \Delta} \right) + \frac{\lambda_i}{\lambda_a} e^{-\lambda_a \Delta} \right] \tag{1}$$

The expression for $P^2_{n_1,n_2}$ is similar, with the sum taken over the flows on path P_2. The transition probabilities can be further approximated as

$$P^j_{n_1,n_2} \approx \frac{C_j}{2(C_1 + C_2)} e^{-(C_j/n_j)\Delta} \tag{2}$$

for $j \in \{0, 1\}$. The approximation is accurate when $\lambda_a \gg \lambda_i$, which is the case if the number of flows is large, and (n_1, n_2) isn't too imbalanced.

Comparing $P^1_{n_1,n_2}$ and $P^2_{n_1,n_2}$ in light of Eq. (2) gives an interesting insight into how flows transition between paths. For a suitably chosen value of Δ, the flows on the path with lower per-flow rate are more likely to change path. This behavior pushes the system towards states where $C_1/n_1 \approx C_2/n_2$,[5] implying good load balancing. Notice that this is exactly the behavior we observed in the previous simulation with TCP flows (Figure 5).

The role of the flowlet timeout (Δ). Achieving the good load balancing behavior above depends on Δ. If Δ is very large, then $P^j_{n_1,n_2} \approx 0$ (no timeouts); if Δ is very small, then $P^j_{n_1,n_2} \approx C_1/2(C_1 + C_2)$. In either case, the transition probabilities don't depend on the rate of the flows and thus the load balancing feature of flowlet switching

[5]The most precise balance condition is $\frac{C_1}{n_1} - \frac{C_2}{n_2} \approx \frac{1}{\Delta} \log(\frac{C_1}{C_2})$.

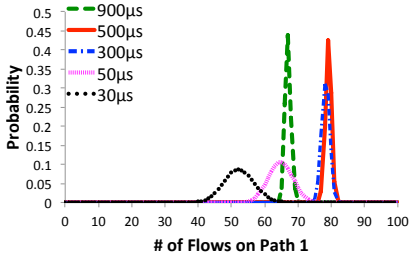


Figure 7: State probability distribution after 10^{11} steps, starting from (50,50), for different values of flowlet timeout.

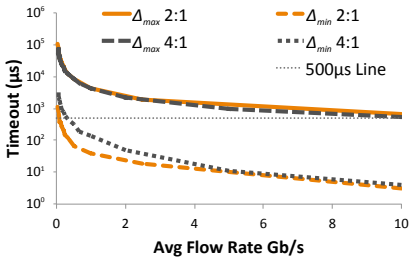


Figure 8: Flowlet timeout range vs. average flow rate.

is lost. This behavior is illustrated in Figure 7, which shows the probability distribution of the number of flows on path 1 for several values of Δ, with $n = 100$ flows, $C_1 = 40$ Gbps, and $C_2 = 10$ Gbps. These plots are obtained by solving for the probability distribution of the Markov chain after 10^{11} steps numerically, starting with an initial state of $(50,50)$.

The ideal operating point in this case is $(80,20)$. The plot shows that for small Δ (30 μs and 50 μs) and large Δ (900 μs), the state distribution is far from ideal. However, for moderate values of Δ (300 μs and 500 μs), the distribution concentrates around the ideal point.

So how should Δ be chosen in practice? This question turns out to be tricky to answer based purely on this model. First, the Poisson assumption does not capture the RTT-dependence and burstiness of TCP flows, both of which impact the inter-packet gap, hence, flowlet timeouts. Second, in picking the flowlet timeout, we must take care to avoid packet reordering (unless other mitigating measures are taken to prevent the adverse effects of reordering [11]).

Nonetheless, the model provides some useful insights for setting Δ. This requires a small tweak to (roughly) model burstiness: we assume that the flows transmit in bursts of b packets, as a Poisson process of rate λ_i/b. Empirically, we find that TCP sends in bursts of roughly $b = 10$ in our simulation setup (§5). Using this value, we run a series of simulations to obtain Δ_{min} and Δ_{max}, the minimum and maximum values of Δ for which the load balancer achieves a near-ideal split of flows across the two paths. In these simulations, we vary the number of flows, and consider two cases with 20/10 Gbps and 40/10 Gbps paths.

Figure 8 shows Δ_{min} and Δ_{max} as a function of the average flow rate in each scenario: $(C_1 + C_2)/n$. There is a clear correlation between the average flow rate and Δ. As the average flow rate increases, Δ needs to be reduced to achieving good load balancing. A flowlet timeout around 500 μs supports the largest range of flow rates. We adopt this value in our experiments (§5).

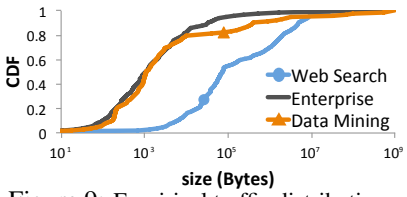


Figure 9: Empirical traffic distributions.

5 Evaluation

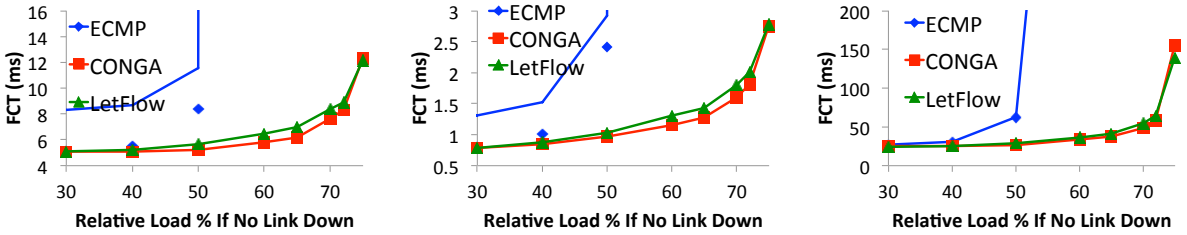
In this section, we evaluate LetFlow's performance with a small hardware testbed (§5.1) as well as large-scale simulations (§5.2). We also study LetFlow's generality and robustness (§5.3).

Schemes compared. In our hardware testbed, we compare LetFlow against ECMP and CONGA [3], a state-of-the-art adaptive load balancing mechanism. In simulations, we compare LetFlow against WCMP [31], CONGA, and Presto*, an idealized variant of Presto [15], a state-of-the-art proactive load balancing scheme that sprays small fixed-sized chunks of data across different paths and uses a reordering buffer at the receivers to put packets of each flow back in order. Presto* employs per-packet load balancing and a perfect reordering buffer that knows exactly which packets have been dropped, and which have been received out-of-order.

Note: Presto* cannot be realized in practice, but it provides the best-case performance for Presto (and other such schemes) and allows us to isolate performance issues due to load balancing from those caused by packet reordering. CONGA has been shown to perform better than MPTCP in scenarios very similar to those that we consider [3]; therefore, we do not compare with MPTCP.

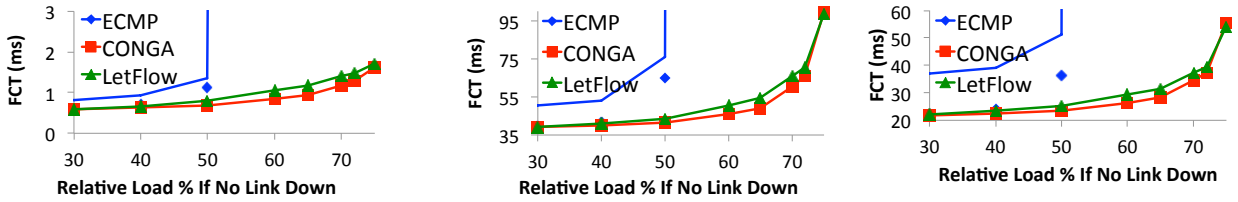
Workloads. We conduct our experiments using realistic workloads based on empirically observed traffic patterns in deployed data centers. We consider the three flow size distributions shown in Figure 9, from (1) production clusters for web search services [4]; (2) a typical large enterprise datacenter [3]; (3) a large cluster dedicated to data mining services [13]. All three distributions are heavy-tailed: a small fraction of the flows contribute most of the traffic. Refer to the papers [4, 3, 13] that report on these distributions for more details.

Metrics. Similar to prior work, we use flow completion time (FCT) as the primary performance metric. In certain experiments, we also consider packet latency and traffic distribution across paths.



(a) Web search workload (b) Enterprise workload (c) Data mining traffic workload

Figure 10: Testbed results: overall average FCT for different workloads on baseline topology with link failure (Fig. 1b).



(a) Avg FCT: small flows (<100 KB) (b) Avg FCT: large flows (>10 MB) (c) 95th percentile FCT

Figure 11: Several FCT statistics for web search workload on baseline topology with link failure.

Summary of results. Our evaluation spans several dimensions: (1) different topologies with varying asymmetry and traffic matrices; (2) different workloads; (3) different congestion control protocols with varying degrees of burstiness; (4) and different network and algorithm settings. We find that:

1. LetFlow achieves very good performance for realistic datacenter workloads in both asymmetric and symmetric topologies. It achieves average FCTs that are significantly better than competing traffic-oblivious schemes such as WCMP and Presto, and only slightly higher than CONGA: within 10-20% in testbed experiments and 2× in simulations with high asymmetry and traffic load.
2. LetFlow is effective in large topologies with high degrees of asymmetry and multi-tier asymmetric topologies, and also handles asymmetries caused by imbalanced and dynamic traffic matrices.
3. LetFlow performs consistently for different transport protocols, including "smooth" congestion control algorithms such as DCTCP [4] and schemes with hardware pacing such as DCQCN [32].
4. LetFlow is also robust across a wide range of buffer size and flowlet timeout settings, though tuning the flowlet timeout can improve its performance.

5.1 Testbed Experiments

Our testbed setup uses a topology that consists of two leaf switches and two spine switches as shown in Figure 1b. Each leaf switch is connected to each spine switch with two 40 Gbps links. We fail one of the links between a leaf and spine switch to create asymmetry, as indicated by the dotted line in Figure 1b. There are 64 servers that are attached to the leaf switches (32 per leaf) with 10 Gbps links. Hence, we have a 2:1 oversubscription at the leaf level, which is typical in modern data centers. The servers have 12 core Intel Xeon X5670 2.93 Ghz CPUs and 96 GB of RAM.

We use a simple client-server program to generate traffic. Each server requests flows according to a Poisson process from randomly chosen servers. The flow size is drawn from one of the three distributions discussed above. All 64 nodes run both client and server processes. To stress the fabric's load balancing, we configure the clients under each leaf to only send to servers under the other leaf, so that all traffic traverses the fabric.

Figure 10 compares the overall average FCT achieved by LetFlow, CONGA and ECMP for the three workloads at different levels of load. We also show a breakdown of the average FCT for small (<100 KB) and large (>10 MB) flows as well as the 95th percentile FCT for the web search workload in Figure 11. (The results for the other workloads are similar.)

In these experiments, we vary the traffic load by changing the arrival rate of flows to achieve different levels of load. The load is shown relative to the bisectional bandwidth without the link failure; i.e., the maximum load that the fabric can support with the link failure is 75% (see Figure 1b). Each data point is the average of ten runs on the testbed.

The results show that ECMP's performance deteriorates quickly as the traffic load increases to 50%. This is not surprising; since ECMP sends half of the traffic through each spine, at 50% load, the $S1 \rightarrow L1$ link (on the path with reduced capacity) is saturated and performance rapidly degrades. LetFlow, on the other hand, performs very well across all workloads and different flow size groups (Figure 11), achieving FCTs that are only 10-20% higher than CONGA in the worst case. This demonstrates the ability of LetFlow to shift traffic

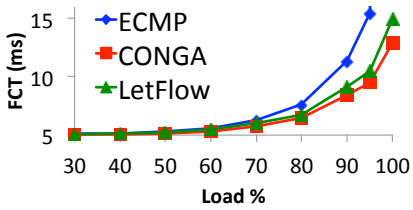


Figure 12: Overall average FCT for web search workload on symmetric testbed experiments.

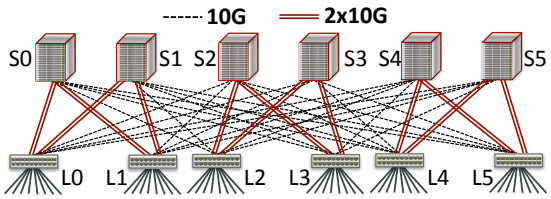


Figure 13: Large-scale topology with high path asymmetry.

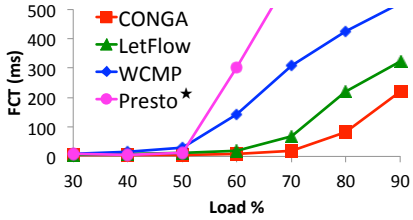


Figure 14: Average FCT comparison (web search workload, large-scale asymmetric topology).

away from the congested link for realistic dynamic traffic loads, without any explicit knowledge of path traffic conditions or complex feedback mechanisms.

Symmetric topology. We repeat these experiments in a symmetric topology without link failures. Figure 12 shows the results for the web search traffic workload. As expected, all three schemes perform similarly when the load is low, but CONGA and LetFlow both outperform ECMP at higher levels of load. Despite its simplicity, LetFlow is within 10% of CONGA at all levels of loads.

5.2 Large Scale Simulations

In this section, we evaluate LetFlow's performance via simulations in larger scale topologies with high degrees of asymmetry and multi-tier asymmetric topologies. We also validate LetFlow's ability to handle challenging traffic scenarios which require different load balancing decisions for traffic destined to different destinations. Our simulations use OMNET++ [28] and the Network Simulation Cradle [17] to port the actual Linux TCP source code (from kernel 2.6.26) as our simulator.

Large-scale topology with high asymmetry. In order to evaluate LetFlow in more complex topologies, we simulate an asymmetric topology with 6 spines and 6 leaf switches, shown in Figure 13. This topology is from the WCMP paper [31]. Spine and leaf switches that belong to the same pod are connected using two 10 Gbps links

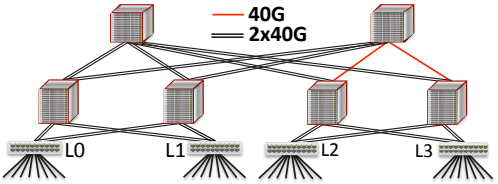


Figure 15: Multi-tier topology with link failure.

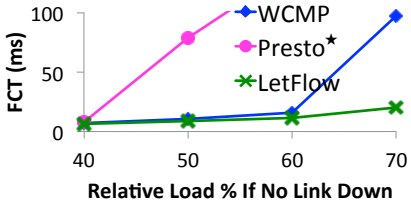


Figure 16: Average FCT (web search workload, asymmetric multi-tier topology).

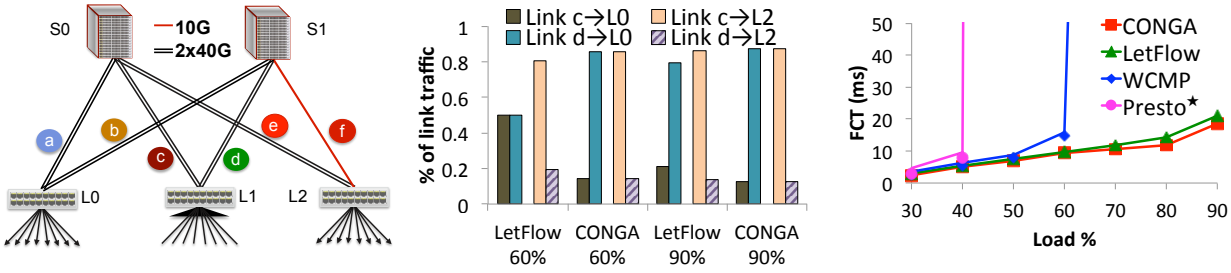
while inter-pod connections use a single 10 Gbps link. Such asymmetric topologies can occur as a result of *imbalanced striping* — a mismatch between switch radices at different levels in a Clos topology [31].

Each leaf is connected to 48 servers. We generate web search traffic uniformly at random from servers at each leaf to servers at the other five leaves. We compare the performance of LetFlow against Presto*, WCMP and CONGA. Presto* and WCMP take into account the path asymmetry by using static weights (based on the topology) to make load balancing decisions, as described in the WCMP [31] and Presto [15] papers.

Figure 14 shows that WCMP and Presto* fail to achieve the performance of CONGA and LetFlow. In this scenario, Presto* cannot finish flows quickly enough and is unstable at 80% and 90% load. CONGA has the best performance overall. But LetFlow also performs quite well: it achieves average FCTs within 50% of CONGA, even at 90% load. This result shows that static topology-dependent weights are inadequate for handling asymmetry with dynamic workloads, and LetFlow is able to balance load well in complex asymmetric topologies.

Multi-tier topology. In order to evaluate LetFlow's scalability, we simulate a 3-tier topology with asymmetry as shown in Figure 15. We aim to validate that LetFlow is able to detect this asymmetry and react accordingly. We focus on the performance of traffic from *L0*-*L1* to *L2*-*L3*.

We do not consider CONGA in this scenario since it was designed for two-tier topologies [3]. We compare LetFlow's performance against WCMP and Presto* for the web search workload. Figure 16 shows the average flow completion time at 40%-70% load. The results show that LetFlow handles asymmetry in the multi-tier topology very well. While Presto* is unstable when the load is larger than 50% and WCMP degrades severely when the load reaches 70%, LetFlow performs similarly to that in the two-tier network.



(a) Topology (b) Traffic split across different paths (c) Avg FCT (web search workload)

Figure 17: Multi-destination scenario with varying asymmetry.

	Destined to $L0$	**Destined to** $L2$
Traffic on *Link c*	11.1%	88.9%
Traffic on *Link d*	88.9%	11.1%

Table 2: Ideal traffic split for multi-destination scenario.

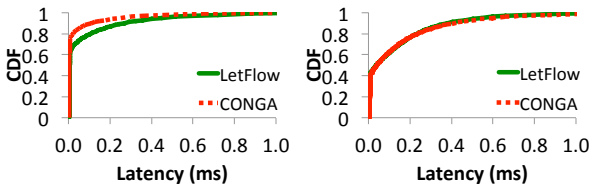
Multiple destinations with varying asymmetry. Load balancing in asymmetric topologies often requires splitting traffic to different destinations differently. For this reason, some load balancing schemes like CONGA [3] take into account the traffic's destination to make load balancing decisions. We now investigate whether LetFlow can handle such scenarios despite making random decisions that are independent of a packet's destination.

We simulate the topology shown in Figure 17a. One leaf switch, $L1$, sends traffic uniformly at random to the other two leaf switches, $L0$ and $L2$, using the web search workload. Note that the two paths from $L1$ to $L0$ are symmetric (both have 80 Gbps of capacity) while the paths between $L1$ to $L2$ are asymmetric: there is only one 10 Gbps link between $S1$ to $L2$. As a result, leaf $L1$ must send more $L1 \rightarrow L2$ traffic via spine $S0$ than $S1$. This in turn creates (dynamic) bandwidth asymmetry for the $L1 \rightarrow L0$ traffic.

The ideal traffic split to balance the maximum load across all fabric links is given in Table 2. We should split the traffic between *Link e* and *Link f* with a ratio of 8:1; i.e., 88.9% of the traffic destined to $L2$ should be carried on *Link c* while 11.1% should be carried on *Link d*. To counter the traffic imbalance on *Link c* and *Link d*, the traffic to $L0$ should ideally be split in the opposite ratio, 1:8, on *Link c* and *Link d*.

Can LetFlow achieve such an ideal behavior? Figure 17b shows the traffic split for LetFlow and CONGA when the total traffic load from $L1$ is 48 Gbps (60%) and 72 Gbps (90%) respectively. We omit ECMP results since it is unstable in this scenario.

We observe that when the traffic load is relatively low (60% for *Link c* and *Link d*), the traffic split with LetFlow for the $L1 \rightarrow L0$ traffic is simply 50%-50%, not ideal. However, notice that at this low load, *Link c* has spare capacity, even with a 50%-50% split for $L1 \rightarrow L0$ traffic. Thus, flowlets destined to $L0$ do not find either link to be particularly better than the other. For traffic destined to $L2$, however, the paths via *Link e* and *Link f*



(a) Symmetric (b) With link failure

Figure 18: Network latency CDF with LetFlow and CONGA on the baseline topology (simulation).

are vastly different. Thus LetFlow's flowlets automatically shift traffic away from the congested link, and the traffic split ratio is 81%/19%, close to ideal.

Now when the load increases to 90%, evenly splitting traffic destined to $L0$ on *Link c* and *Link d* would cause *Link c* to be heavily congested. Interestingly, in this case, LetFlow avoids congestion and moves more $L1 \rightarrow L0$ traffic to *Link d* and achieves close to the ideal ratio.

LetFlow's good performance is also evident from the average FCT in this scenario, shown in Figure 17c. LetFlow achieves similar performance as CONGA (at most 10% worst), which achieves close-to-ideal traffic splits for both $L0$- and $L2$-bound traffic.

Impact on Latency. Finally, we compare LetFlow and CONGA in terms of their impact on fabric latency. We omit the results for ECMP and Presto* which perform much worse in asymmetric topologies, as to examine closely the differences between CONGA and LetFlow. We measure a packet's fabric latency by timestamping it when it enters the ingress leaf switch and when it enters the egress leaf switch. This allows us to measure the fabric latency without including the latency incurred at the egress switch, which is not influenced by load balancing.

Figure 18 shows the CDF of fabric latency for the web search workload at 60% average load in the topology shown in Figure 1b, with and without asymmetry. In the heavily congested asymmetric scenario, LetFlow has similar fabric latency to CONGA. In the symmetric topology, however, CONGA achieves lower fabric latency. The reason is that CONGA can detect subtle traffic imbalances on the two paths and proactively balance load by choosing the path that is least congested. LetFlow, however, is reactive and needs to cause congestion

(e.g. packet loss, increased delay) for TCP to "kick in" and the flowlet sizes to change. Hence, its reaction time is slower than CONGA. This distinction is less relevant in the asymmetric scenario where congestion is high and even CONGA can't avoid increasing queuing delay.

5.3 Robustness

We have shown that LetFlow can effectively load balance traffic under various conditions. In this section, we evaluate LetFlow's robustness to different factors, particularly, different transport protocols, flowlet time timeout periods, and buffer sizes.

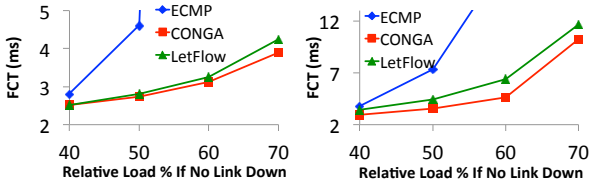
Different transport protocols. As explained in §4, LetFlow's behavior depends on the nature of inter-packet gaps. A natural question is thus how effective is LetFlow for other transport protocols that have different burstiness characteristics than TCP? To answer this question, we repeat the previous experiment for two new transports: DCTCP [4] and DCQCN [32].

DCTCP is interesting to consider because it adjusts its window size much more smoothly than TCP and largely avoids packet drops that can immediately cause flowlet timeouts. To use DCTCP, we enable ECN marking at the switches and set the marking threshold to be 100 KB. Since DCTCP reduces queuing delay, we also set the flowlet table timeout period to 50 μs. We discuss the impact of the flowlet timeout parameter further in the following section. Figure 19a compares the overall average flow completion time using CONGA and LetFlow (with DCTCP as the transport). Similar to the case with TCP, we observe that LetFlow is able to achieve performance within 10% of CONGA in all cases.

Next, we repeat the experiment for DCQCN [32], a recent end-to-end congestion control protocol designed for RDMA environments. DCQCN operates in the NICs to throttle flows that experience congestion via hardware rate-limiters. In addition, DCQCN employs Layer 2 Priority Flow Control (PFC) to ensure lossless operations.

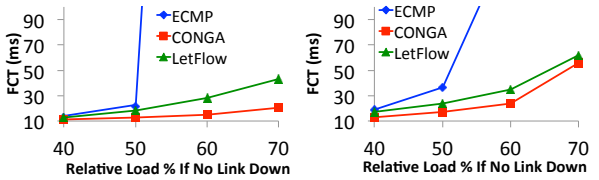
We use the same flowlet table timeout period of 50 μs as in DCTCP's simulation. Figure 19b shows that, while ECMP's performance becomes unstable at loads above 50%, both CONGA and LetFlow can maintain stability by moving enough traffic away from the congested path. LetFlow achieves an average FCT within 47% of CONGA. The larger gap relative to CONGA compared to TCP-based transports is not surprising since DCQCN generates very smooth, perfectly paced traffic, which reduces flowlet opportunities. Nonetheless, as our analysis (§4) predicts, LetFlow still balances load quite effectively, because DCQCN adapts the flow rates to traffic conditions on their paths.

At the 95th percentile, LetFlow's FCT is roughly within 2× of CONGA (and still much better than ECMP)



(a) DCTCP (b) DCQCN

Figure 19: Overall average FCT with different transport protocols (web search workload, baseline topology).



(a) DCTCP (b) DCQCN

Figure 20: 95th percentile FCT with different transport protocols (web search workload, baseline topology).

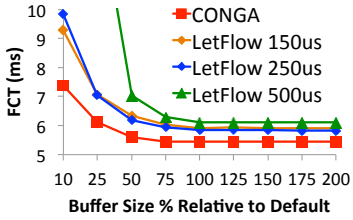


Figure 21: Effect of buffer size and flowlet timeout period on FCT (web search workload at 60% load, baseline topology).

for DCTCP, as shown in Figure 20a. The degradation for DCQCN is approximately 1.5× of CONGA (Figure 20b). This degradation at the tail is not surprising, since unlike CONGA which proactively balances load, LetFlow is reactive and its decisions are stochastic.

Varying flowlet timeout. The timeout period of the flowlet table is fundamental to LetFlow: it determines whether flowlets occur frequently enough to balance traffic. The analysis in §4 provides a framework for choosing the correct flowlet timeout period (see Figure 8). In Figure 21, we show how FCT varies with different values of the flowlet timeout (150 μs, 250 μs and 500 μs) and different buffer sizes in the testbed experiment. These experiments use the web search workload at 60% load and TCP New Reno as the transport protocol. The buffer size is shown relatively to the maximum buffer size (10 MB) of the switch.

When the buffer size is not too small, different timeout periods do not change LetFlow's performance. Since all three values are within the minimum and maximum bounds in Figure 8, we expect all three to perform well. However, for smaller buffer sizes, the performance improves with lower values of the flowlet timeout. The reason is that smaller buffer sizes result in smaller RTTs, which makes TCP flows less bursty. Hence, the burst parameter, b, in §4.2 (set to 10 by default) needs to be

reduced for the model to be accurate.

Varying buffer size. We also evaluate LetFlow's performance for different buffer sizes. As shown in Figure 21, the FCT suffers when the buffer is very small (e.g., 10% of the max) and gradually improves as the buffer size increases to 75% of the maximum size. Beyond this point, the FCT is flat. LetFlow exhibits similar behavior to CONGA in terms of dependency on buffer size.

6 Related Work

We briefly discuss related work, particularly for datacenter networks, that has inspired and informed our design.

Motivated by the drawbacks of ECMP's flow-based load balancing [16], several papers propose more fine-grained mechanisms, including flowlets [20], spatial splitting based on TCP sequence numbers [24], per-packet round robin [9], and load balancing at the level of TCP Segmentation Offload (TSO) units [15]. Our results show that such schemes cannot handle asymmetry well without path congestion information.

Some schemes have proposed topology-dependent weighing of paths as a way of dealing with asymmetry. WCMP [31] adds weights to ECMP in commodity switches, while Presto [15] implements weights at the end-hosts. While weights can help in some scenarios, static weights are generally sub-optimal with asymmetry, particularly for dynamic traffic workloads. LetFlow improves resilience to asymmetry even without weights, but can also benefit from better weighing of paths based on the topology or coarse estimates of traffic demands (e.g., see experiment in Figure 2).

DeTail [30] propose per-packet adaptive load balancing, but requires priority flow control for hop-by-hop congestion feedback. Other dynamic load balancers like MPTCP [23], TeXCP [19], CONGA [3], and HULA [21] load balance traffic based on path-wise congestion metrics. LetFlow is significantly simpler compared to these designs as its decisions are local and purely at random.

The closest prior scheme to LetFlow is FlowBender [18]. Like LetFlow, FlowBender randomly reroutes flows, but relies on explicit path congestion signals such as per-flow ECN marks or TCP Retransmission Timeout (RTO). LetFlow does not need any explicit congestion signals or TCP-specific mechanisms like ECN, and can thus support different transport protocols more easily.

Centralized load balancing schemes such as Hedera [2] and MicroTE [8] tend to be slow [23] and also have difficulties handling traffic volatility. Fastpass [22] is a centralized arbiter which can achieve near-ideal load balancing by determining the transmission time and path for every packet, but scaling a centralized per-packet arbiter to large-scale datacenters is very challenging.

7 Final Remarks

Let the flowlets flow!

The main thesis of this paper is that flowlet switching is a powerful technique for simple, resilient load balancing in the presence of network asymmetry. The ability of flowlets to automatically change size based on real-time traffic conditions on their path enables them to effectively shift traffic away from congested paths, without the need for explicit path congestion information or complex feedback mechanisms.

We designed LetFlow as an extreme example of this approach. LetFlow simply picks a path uniformly at random for each flowlet, leaving it to the flowlets to do the rest. Through extensive evaluation in a real hardware testbed and large-scale simulations, we showed that LetFlow has comparable performance to CONGA [3], e.g., achieving average flow completion times within 10-20% of CONGA in our testbed experiments and 2× of CONGA in challenging simulated scenarios.

LetFlow is not an optimal solution or the only way to use flowlet switching. By its reactive and stochastic nature, LetFlow cannot prevent short-time-scale traffic imbalances that can increase queuing delay. Also, in symmetric topologies, schemes that balance load proactively at a more fine-grained level (e.g., packets or small-sized chunks [15]) would perform better.

LetFlow is, however, a significant improvement over ECMP and can be deployed today to greatly improve resilience to asymmetry. It is trivial to implement in hardware, does not require any changes to end-hosts, and is incrementally deployable. Even if only some switches use LetFlow (and others use ECMP or some other mechanism), flowlets can adjust to bandwidth asymmetries and improve performance for all traffic.

Finally, LetFlow is an instance of a more general approach to load balancing that randomly reroutes flows with a probability that decreases as a function of the flow's rate. Our results show that this simple approach works well in multi-rooted tree topologies. Modeling this approach for general topologies and formally analyzing its stability and convergence behavior is an interesting avenue for future work.

Acknowledgements

We are grateful to our shephard, Thomas Anderson, the anonymous NSDI reviewers, Akshay Narayan, and Srinivas Narayana for their valuable comments that greatly improved the clarity of the paper. We are also thankful to Edouard Bugnion, Peter Newman, Laura Sharpless, and Ramanan Vaidyanathan for many fruitful discussions. This work was funded in part by NSF grants CNS-1617702 and CNS-1563826, and a gift from the Cisco Research Center.

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[3] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 503–514, New York, NY, USA, 2014. ACM.

[4] M. Alizadeh et al. Data center TCP (DCTCP). In *SIGCOMM*, 2010.

[5] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing Router Buffers. In *SIGCOMM*, 2004.

[6] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *SIGCOMM*, 2010.

[7] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. *SIGCOMM Comput. Commun. Rev.*, 40(1):92–99, Jan. 2010.

[8] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *CoNEXT*, 2011.

[9] J. Cao et al. Per-packet Load-balanced, Low-latency Routing for Clos-based Data Center Networks. In *CoNEXT*, 2013.

[10] A. Dixit, P. Prakash, Y. Hu, and R. Kompella. On the impact of packet spraying in data center networks. In *INFOCOM*, 2013.

[11] Y. Geng, V. Jeyakumar, A. Kabbani, and M. Alizadeh. Juggler: a practical reordering resilient network stack for datacenters. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 20. ACM, 2016.

[12] P. Gill, N. Jain, and N. Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *SIGCOMM*, 2011.

[13] A. Greenberg et al. VL2: a scalable and flexible data center network. In *SIGCOMM*, 2009.

[14] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *SIGCOMM*, 2009.

[15] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. Presto: Edge-based load balancing for fast datacenter networks. 2015.

[16] C. Hopps. Analysis of an equal-cost multi-path algorithm, 2000.

[17] S. Jansen and A. McGregor. Performance, Validation and Testing with the Network Simulation Cradle. In *MASCOTS*, 2006.

[18] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene. Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 149–160. ACM, 2014.

[19] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the Tightrope: Responsive Yet Stable Traffic Engineering. In *SIGCOMM*, 2005.

[20] S. Kandula, D. Katabi, S. Sinha, and A. Berger. Dynamic Load Balancing Without Packet Reordering. *SIGCOMM Comput. Commun. Rev.*, 37(2):51–62, Mar. 2007.

[21] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. Hula: Scalable load balancing using programmable data planes. In *Proc. ACM Symposium on SDN Research*, 2016.

[22] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized zero-queue datacenter network. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 307–318. ACM, 2014.

[23] C. Raiciu et al. Improving datacenter performance and robustness with multipath tcp. In *SIGCOMM*, 2011.

[24] S. Sen, D. Shue, S. Ihm, and M. J. Freedman. Scalable, Optimal Flow Routing in Datacenters via Local Link Balancing. In *CoNEXT*, 2013.

[25] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hoelzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Sigcomm '15*, 2015.

[26] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking Data Centers Randomly. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[27] S. Sinha, S. Kandula, and D. Katabi. Harnessing TCPs Burstiness using Flowlet Switching. In *3rd ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, San Diego, CA, November 2004.

[28] A. Varga et al. The OMNeT++ discrete event simulation system. In *ESM*, 2001.

[29] P. Wang, H. Xu, Z. Niu, D. Han, and Y. Xiong. Expeditus: Congestion-aware load balancing in clos data center networks. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 442–455, New York, NY, USA, 2016. ACM.

[30] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. H. Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *SIGCOMM*, 2012.

[31] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat. Wcmp: weighted cost multipathing for improved fairness in data centers. In *Proceedings of the Ninth European Conference on Computer Systems*, page 5. ACM, 2014.

[32] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale rdma deployments. *SIGCOMM Comput. Commun. Rev.*, 45(4):523–536, Aug. 2015.

A Derivation of Equation (1)

Suppose n flows share a link, and flow i transmits packets as a Poisson process with a rate λ_i, independent of the other flows. Let $\lambda_a = \sum_{i=1}^{n} \lambda_i$ be the aggregate packet arrival rate. Notice that the packet arrival process for all flows besides flow i is Poisson with rate $\lambda_a - \lambda_i$.

Consider an arbitrary time instant t. Without loss of generality, assume that flow i is the next flow to incur a flowlet timeout after t. Let τ_i^{-1} be the time interval between flow i's last packet arrival and t, and τ_i be the time interval between t and flow i's next packet arrival. Also, let $\tau_{a\setminus i}$ be the time interval until the next packet arrival from any flow other than flow i. Figure 22 shows these quantities.

The probability that flow i incurs the next flowlet timeout, $\mathbb{P}_i$, is the joint probability that the next packet arrival after time t is from flow i, and its inter-packet gap, $\tau_i^{-1}+\tau_i$, is larger than the flowlet timeout period Δ:

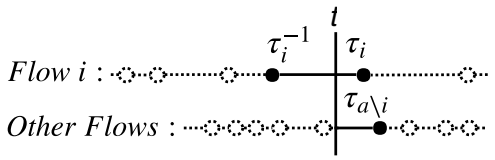


Figure 22: Packet arrival processes for flow i and other flows, and their relationship to each other

$$\begin{aligned}\mathbb{P}_i &= \mathbb{P}\left\{\tau_i < \tau_{a\setminus i}, \tau_i + \tau_i^{-1} > \Delta\right\} \\ &= \int_0^\infty \lambda_i e^{-\lambda_i t}\mathbb{P}\left\{\tau_{a\setminus i} > t, \tau_i^{-1} > \Delta - t\right\} dt. \quad (3)\end{aligned}$$

Since packet arrivals of different flows are independent, we have:

$$\mathbb{P}\left\{\tau_{a\setminus i} > t, \tau_i^{-1} > \Delta - t\right\} = \mathbb{P}\left\{\tau_{a\setminus i} > t\right\}\mathbb{P}\left\{\tau_i^{-1} > \Delta - t\right\}.$$

Also, it follows from standard properties of Poisson processes that

$$\begin{aligned}\mathbb{P}\left\{\tau_{a\setminus i} > t\right\} &= e^{-(\lambda_a-\lambda_i)t} \\ \mathbb{P}\left\{\tau_i^{-1} > \Delta - t\right\} &= \begin{cases} e^{-\lambda_i(\Delta-t)} & t \le \Delta \\ 1 & t \ge \Delta. \end{cases} \quad (4)\end{aligned}$$

Therefore, we obtain

$$\begin{aligned}\mathbb{P}_i &= \int_0^\Delta \lambda_i e^{-\lambda_i\Delta} e^{-(\lambda_a-\lambda_i)t} dt + \int_\Delta^\infty \lambda_i e^{-\lambda_a t} dt \\ &= \frac{\lambda_i}{\lambda_a - \lambda_i}\left(e^{-\lambda_i\Delta} - e^{-\lambda_a\Delta}\right) + \frac{\lambda_i}{\lambda_a} e^{-\lambda_a\Delta}. \quad (5)\end{aligned}$$

Now, assume that there are two paths P_1 and P_2, and let n_1 and n_2 denote the number of flows on each path. Also, assume that the total arrival rate on both paths is given by λ_a. Following a flowlet timeout, the flow with the timeout is assigned to a random path. It is not difficult to see that (n_1, n_2) forms a Markov chain (see Figure 6).

Let $P^1_{n_1,n_2}$ and $P^2_{n_1,n_2}$ be the transition probabilities from (n_1, n_2) to $(n_1 - 1,\ n_2 + 1)$ and $(n_1 + 1,\ n_2 - 1)$ respectively. To derive $P^1_{n_1,n_2}$, notice that the probability that the next flowlet timeout occurs for one of the flows on path P_1 is given by $\sum_{i\in P_1}\mathbb{P}_i$, where the notation $i \in P_1$ indicates that the sum is over the flows on path P_1, and $\mathbb{P}_i$ is given by the expression in Eq. (5). The flow that times out will change paths with probability $1/2$. Therefore:

$$P^1_{n_1,n_2} = \frac{1}{2}\sum_{i\in P_1}\left[\frac{\lambda_i}{\lambda_a - \lambda_i}\left(e^{-\lambda_i\Delta} - e^{-\lambda_a\Delta}\right) + \frac{\lambda_i}{\lambda_a}e^{-\lambda_a\Delta}\right], \quad (6)$$

which is Equation (1) in §4. The derivation for $P^2_{n_1,n_2}$ is similar.

Flowtune: Flowlet Control for Datacenter Networks

Jonathan Perry, Hari Balakrishnan and Devavrat Shah
Computer Science and Artificial Intelligence Lab, M.I.T.
Email: {yonch, hari, devavrat}@mit.edu

Abstract

Rapid convergence to a desired allocation of network resources to endpoint traffic is a difficult problem. The reason is that congestion control decisions are distributed across the endpoints, which vary their offered load in response to changes in application demand and network feedback on a packet-by-packet basis. We propose a different approach for datacenter networks, *flowlet control*, in which congestion control decisions are made at the granularity of a flowlet, not a packet. With flowlet control, allocations have to change only when flowlets arrive or leave. We have implemented this idea in a system called Flowtune using a centralized allocator that receives flowlet start and end notifications from endpoints. The allocator computes optimal rates using a new, fast method for network utility maximization, and updates endpoint congestion-control parameters. Experiments show that Flowtune outperforms DCTCP, pFabric, sfqCoDel, and XCP on tail packet delays in various settings, converging to optimal rates within a few packets rather than over several RTTs. Benchmarks on an EC2 deployment show a fairer rate allocation than Linux's Cubic. A data aggregation benchmark shows 1.61× lower p95 coflow completion time.

1 Introduction

Over the past thirty years, network congestion control schemes—whether distributed [24, 8, 21, 20, 39] or centralized [33], whether end-to-end or with switch support [12, 16, 17, 35, 26, 38, 32], and whether in the wide-area Internet [13, 42] or in low-latency datacenters [2, 3, 4, 22, 30]—have operated at the granularity of individual packets. Endpoints transmit data at a rate (window) that changes from packet to packet.

Packet-level network resource allocation has become the de facto standard approach to the problem of determining the rates of each flow in a network. By contrast, *if* it were possible to somehow determine optimal rates for a set of flows sharing a network, then those rates would have to change *only* when new flows arrive or flows leave the system. Avoiding packet-level rate fluctuations could help achieve *fast convergence* to optimal rates.

For this reason, in this paper, we adopt the position that a *flowlet*, and not a packet, is a better granularity for congestion control. By "flowlet", we mean a batch of packets that are backlogged at a sender; a flowlet ends when there is a threshold amount of time during which a sender's queue is empty. Our idea is to compute optimal rates for a set of active flowlets and to update those rates dynamically as flowlets enter and leave the network.[1]

We have developed these ideas in a system called *Flowtune*. It is targeted at datacenter environments, although it may also be used in enterprise and carrier networks, but is not intended for use in the wide-area Internet.

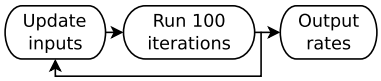
In datacenters, fast convergence of allocations is critical, as flowlets tend to be short (one study shows that the majority of flows are under 10 packets [9]) and link capacities are large (40 Gbits/s and increasing). If it takes longer than, say, 40 μs to converge to the right rate, then most flowlets will have already finished. Most current approaches use distributed congestion control, and generally take multiple RTTs to converge. By contrast, Flowtune uses a centralized rate allocator.

Computing the optimal rates is difficult because even one flowlet arriving or leaving could, in general, cause updates to the rates of many existing flows. Flows that share a bottleneck with the new or ending flow would change their rates. But, in addition, if some of these flows slow down, other flows elsewhere in the network might be able to speed up, and so on. The effects can cascade.

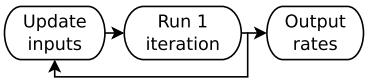
To solve this problem in a scalable way, Flowtune uses the *network utility maximization* (NUM) framework, previously developed to analyze distributed congestion control protocols [27, 29]. In Flowtune, network operators specify an explicit objective. We introduce a new method, termed *Newton-Exact-Diagonal* (NED), that converges quickly to the allocation that maximizes the specified utility (§2).

Flowtune can achieve a variety of desirable objectives. In this paper, we focus on proportional fairness, i.e.,

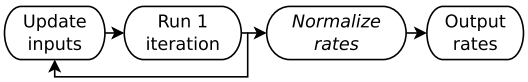
[1] Long-lived flows that send intermittently generate multiple flowlets.



(a) The strawman solution runs the optimizer until it converges, but then system is slow to react to updates of flowlet start/end.



(b) Performing just one iteration reduces reaction time, but results in over-allocation while the algorithm converges.



(c) Flowtune adds a normalization step that eliminates link over-allocation while achieving 99.7% of optimal throughput.

Figure 1: Motivation for the allocator architecture.

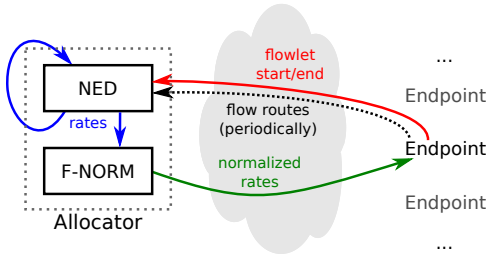


Figure 2: Flowtune components. Endpoints send notifications of new flowlets starting and current flowlets finishing to the allocator. The NED optimizer computes rates, which F-NORM then normalizes and sends back to Endpoints; endpoints adjust sending rates accordingly.

$\max \sum_i U(x_i)$, where $U(x_i) = \log x_i$, and x_i is the throughput of flowlet i. In general, however, NED supports any objective where the utility is a function of the flow's allocated rate.[2]

Endpoints report to the allocator whenever a flowlet starts or ends. Rather than waiting for the optimization to converge before setting rates (Figure 1a), the allocator continuously updates its set of active flowlets and sets intermediate rates between iterations of the optimizer (Figure 1b). These intermediate rates may temporarily exceed the capacity of some links, causing queuing delays. To reduce queuing, Flowtune uses a *rate normalizer* (F-NORM, §3) to scale-down the computed values (Figure 1c).

The normalizer's results are sent to the endpoints. Endpoints transmit according to these rates (they are trusted, similar to trust in TCP transmissions today). Figure 2 shows the system components and their interactions.

[2]Under some requirements of utility functions, discussed in §2.

Flowtune does not select flow paths, but rather works given the paths the network selects for each flow (§6).

A scalable implementation of the optimization algorithm on CPUs would run in parallel on multiple cores. Unfortunately, straightforward implementations are slowed by expensive cache-coherence traffic. We propose a partitioning of flows to cores where each core only interacts with a small set of links. Each core has copies of link state it needs. Before manipulating link state, the algorithm aggregates all modified copies of link state to authoritative copies. Afterwards, the algorithm distributes copies back to the cores (§4). On 40 Gbits/s links, this scheme allows our implementation to allocate 15.36 Tbit/s in 8.29 on 4 Nehalem cores, up to 184 Tbit/s in 30.71 µs on 64 Nehalem cores (§5.2).

We implemented Flowtune in a Linux kernel module and a C++ allocator that implements the multi-core NED algorithm and uses kernel-bypass for NIC access. The system enforces rate allocations on unmodified Linux applications. We deployed Flowtune on Amazon Web Services instances; experiments show the servers are able to achieve their fair share of available network resources, with much better fairness than the Linux baseline (which uses Cubic). Flowtune reduced the 95th percentile (p95) of coflow completion times [10] by 1.61× on a data aggregation benchmark.

Simulation results show that Flowtune out-performs distributed congestion control methods like DCTCP, pFabric, Cubic-over-sfqCoDel, and XCP on metrics of interest like the convergence time and the p99 of the flow completion time (FCT).

Compared with the centralized arbitration in Fastpass [33], Flowtune offers similar fast convergence, but handles 10.4× traffic per core and utilizes 8× more cores, for an improvement in throughput by a factor of 83.2.

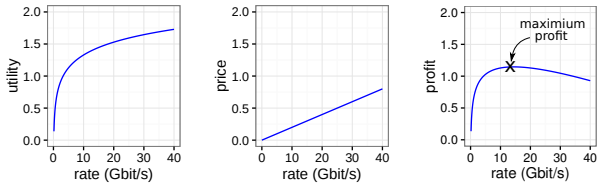
2 Rate Allocation in Flowtune

Solving an explicit optimization problem allows Flowtune to converge rapidly to the desired optimal solution. To our knowledge, NED is the first NUM scheme designed specifically for fast convergence in the centralized setting.

2.1 Intuition

The Flowtune rate allocator chooses flow rates by introducing link prices. The allocator adjusts link prices based on demand, increasing the price when the demand is high and decreasing it when demand is low.

Figure 3 shows how the allocator chooses a flow's rate given link prices. It takes the flow's utility function (3a). Then, it determines the flow's price per unit bandwidth, which is the sum of prices on links it traverses (3b). The utility minus price determines the profit (3c); the allocator chooses the rate that maximizes the flow's profit.



(a) Each flow has a utility function.

(b) Link prices determine the cost per unit rate.

(c) The rate maximizes profit, i.e., utility minus price.

Figure 3: Illustration of how NUM chooses a flow rate.

Intuitively, prices should be adjusted strongly when demand is far from capacity, and gently when it is close. But by exactly how much should an algorithm adjust prices in a given setting?

The exact recipe for adjusting prices is the key differentiator among different algorithms to solve NUM. Simplistic methods can adjust prices too gently and be slow to converge, or adjust prices too aggressively and cause wild fluctuations in rates, or not even converge.

NED converges quickly because it adjusts prices not only using the difference between demand and capacity, but also using an estimate of how strongly rates will change for a given change in price.

2.2 The NUM framework

The following table summarizes the notation used in this paper:

L	Set of all links	L(s)	Links traversed by flow s
S	Set of all flows	$S(\ell)$	Flows that traverse link ℓ
p_ℓ	Price of link ℓ	c_ℓ	Capacity of link ℓ
x_s	Rate of flow s	$U_s(x)$	Utility of flow s
G_ℓ	By how much link ℓ is over-allocated		
$H_{\ell\ell}$	How much flow rates on ℓ react to a change in p_ℓ		

The goal is to allocate rates to all flows subject to network resource constraints: for each link $\ell \in L$,

$$\sum_{s \in S(\ell)} x_s \leq c_\ell. \tag{1}$$

Note that in general many allocations satisfy this constraint. Among these, NUM proposes that we should choose the one that maximizes the overall network utility, $\sum_{s \in S} U_s(x_s)$. Thus, the rate allocation should be the solution to the following optimization problem:

$$\max \sum_s U_s(x_s) \tag{2}$$
$$\text{over } x_s \geq 0, \text{ for all } s \in S,$$
$$\text{subject to (1).}$$

Solving NUM using prices. The capacity constraints in (1) make it hard to solve the optimization problem directly. Kelly's approach to solving NUM [27] is to use Lagrange multipliers, which replace the hard capacity constraints with a "utility penalty" for exceeding capacities. This is done by introducing prices for links.

With prices, each flow selfishly optimizes its own profit, i.e., chooses a rate such that its utility, minus the price it pays per unit bandwidth on the links it traverses, is maximized. Although each flow is selfish, the system still converges to a global optimum because prices force flows to make globally responsible rate selections.[3]

An important quantity to consider when adjusting prices is by how much each link is over-allocated, i.e., $G_\ell = (\sum_{s \in S(\ell)} x_s) - c_\ell$. If $G_\ell > 0$, the link price should increase; if $G_\ell < 0$ it should decrease.

Appendix A outlines why price duality works. Related NUM algorithms are discussed in §7 and Appendix B.

2.3 The NED algorithm

The key observation in NED that enables its fast convergence is that given the utility functions, it is possible to directly compute (1) flow rates given prices, and (2) how strongly flows on a link ℓ will react to a change in that link's price, which we denote $H_{\ell\ell}$.[4]

Direct computation of values eliminates the need to measure the network, and thus greatly speeds up algorithm iterations. In contrast to the full Newton's method, prices updates based on the diagonal can be computed quickly enough on CPUs for sizeable topologies. This results in the update rule:

$$p_\ell \leftarrow p_\ell + \gamma G_\ell H_{\ell\ell}^{-1}.$$

We note that the ability to directly compute $H_{\ell\ell}$ originates from the ability to reliably obtain the above values, not the centralization of the allocator. When the endpoints are trusted, a distributed implementation of NED can use the endpoints to compute and supply these values.

Algorithm 1 shows Flowtune's Newton-Exact-Diagonal (NED) rate allocation algorithm. In Flowtune, the initialization of prices happens only once, when the system first starts. The allocator starts without any flows, and link prices are all set to 1. When flows arrive, their initial rates are computed using current prices.

Choice of utility function. NED admits any utility function U_s that is strictly concave, differentiable, and monotonically increasing. For example, the logarithmic utility function, $U(x) = w \log x$ (for some weight $w > 0$), will optimize weighted proportional fairness [27].

[3] We discuss the requirements for convergence further below.

[4] H is in fact the Hessian; NED computes the Hessian's diagonal, $H_{\ell\ell}$. The Hessian's diagnoal is where NED gets its name.

Algorithm 1 Single iteration of Newton-Exact-Diagonal

NED updates rates $\mathbf{x} = (x_s)$ given prices $\mathbf{p} = (p_\ell)$ ("rate update" step). Then, in the next step of the iteration ("price update"), it uses the updated rates to update the prices.

Rate update. Given prices $\mathbf{p} = (p_\ell)$, for each flow $s \in S$, update the rate:

$$x_s = x_s(\mathbf{p}) = \left(U_s'\right)^{-1}\Big(\sum_{\ell \in L(s)} p_\ell\Big). \tag{3}$$

For example, if $U_s(x) = w \log x$, then $x_s = \frac{w}{\sum_{\ell \in L(s)} p_\ell}$.

Price update. Given updated rates $\mathbf{x} = \mathbf{x}(\mathbf{p}) = (x_s(\mathbf{p}))$ as described above, update the price of each link $\ell \in L$:

$$p_\ell \leftarrow \max\left(0, p_\ell - \gamma H_{\ell\ell}^{-1} G_\ell\right), \tag{4}$$

where $\gamma > 0$ is a fixed algorithm parameter (e.g. $\gamma = 1$), $G_\ell = \left(\sum_{s \in S(\ell)} x_s\right) - c_\ell$, $H_{\ell\ell} = \sum_{s \in S(\ell)} \frac{\partial x_s(\mathbf{p})}{\partial p_\ell}$.
From (3), $\frac{\partial x_s(\mathbf{p})}{\partial p_\ell} = \left(\left(U_s'\right)^{-1}\right)'\left(\sum_{m \in L(s)} p_m\right)$.

3 Rate normalization

The optimizer works in an online setting: when the set of flows changes, the optimizer does not start afresh, but instead updates the previous prices with the new flow configuration. While the prices re-converge, there are momentary spikes in throughput on some links. Spikes occur because when one link price drops, flows on the link increase their rates and cause higher, over-allocated demand on other links (shown in §3).

Normally, allocating rates above link capacity results in queuing. The centralized optimizer can avoid queuing and its added latency by normalizing allocated rates to link capacities. We propose two schemes for normalization: *uniform normalization* and *flow normalization*. For simplicity, the remainder of this section assumes all links are allocated non-zero throughput; it is straightforward to avoid division by zero in the general case.

Uniform normalization (U-NORM): U-NORM scales the rates of all flows by a factor such that the most congested link will operate at its capacity. U-NORM first computes for each link the ratio of the link's allocation to its capacity $r_\ell = \sum_{s \in S(\ell)} x_s / c_\ell$. The most over-congested link has the ratio $r^\star = \max_{\ell \in L} r_\ell$; all flows are scaled using this ratio:

$$\bar{x}_s = \frac{x_s}{r^\star}. \tag{5}$$

The benefits of uniform scaling of all flows by the same constant are the scheme's simplicity, and that it preserves the relative sizes of flows; for utility functions of the form $w \log x_s$, this preserves the fairness of allocation. However, as shown in §3, uniform scaling tends to scale down flows too much, reducing total network throughput.

Flow normalization (F-NORM) Per-flow normalization scales each flow by the factor of its most congested link. This scales down all flows passing through a link ℓ by *at least* a factor of r_ℓ, which guarantees the rates through the link are at most the link capacity. Formally, F-NORM sets

$$\bar{x}_s = \frac{x_s}{\max_{\ell \in L(s)} r_\ell}. \tag{6}$$

F-NORM requires per-flow work to calculate normalization factors, and does not preserve relative flow rates, but a few over-allocated links do not hurt the entire network's throughput. Instead, only the flows traversing congested links are scaled down.

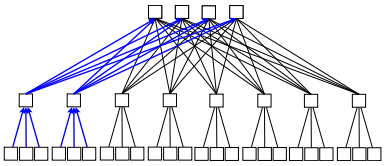
We note that the normalization of flow rates follows a similar structure to NED but instead of prices, the algorithm computes normalization factors. This allows F-NORM to reuse the multi-core design of NED, as described in §4.

4 Scalability

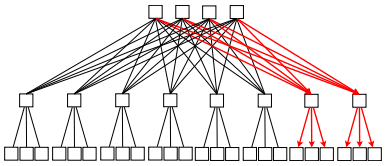
The allocator scales by working on multiple cores on one of more machines. Our design and implementation focuses on optimizing 2-stage Clos networks such as a Facebook fabric pod [5] or a Google Jupiter aggregation block [36], the latter consisting of 6,144 servers in 128 racks. We believe the techniques could be generalized to 3-stage topologies, but demonstrating that is outside the scope of this paper.

A strawman multiprocessor algorithm, which arbitrarily distributes flows to different processors, will perform poorly because NED uses flow state to update link state when it computes aggregate link rates from flow rates: updates to a link from flows on different processors will cause significant cache-coherence traffic, slowing down the computation.

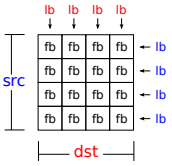
Reducing concurrent updates. Now consider an algorithm that distributes flows to processors based on source rack. This algorithm is still likely to be sub-optimal: flows from many source racks can all update links to the same destination, again resulting in expensive coherence traffic. However, this grouping has the property that all updates to links connecting servers→ToR switches and ToR→aggregation switches (i.e., going *up* the topology) are only performed by the processor responsible for the source rack. A similar grouping by destination rack has locality in links going *down* the topology. Flowtune uses this observation for its multi-processor implementation.



(a) Upward links from a set of racks form an *upward LinkBlock*. Only flows originating from these racks update this LinkBlock.

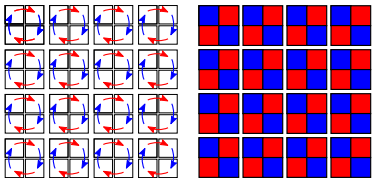


(b) Downward links towards a set of racks form a *downward LinkBlock*. Only flows destined to these racks update this LinkBlock.

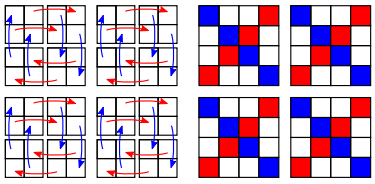


(c) Flows are partitioned by source and destination into FlowBlocks, each updating an upward (blue) and a downward (red) LinkBlock.

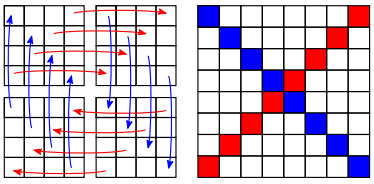
Figure 4: Partitioning of network state.



(a) Step 1: 2x2 processors.



(b) Step 2: 4x4 processors.



(c) Step 3: 8x8 processors.

Figure 5: Aggregation of per-processor LinkBlock state in a 64-processor setup. At the end of step m, blocks of $2m \times 2m$ processors have aggregated upward LinkBlocks on the main diagonal, and downward LinkBlocks on the secondary diagonal.

Figure 4 shows the partitioning of flows and links into FlowBlocks and LinkBlocks. Groups of network racks form *blocks* (two racks per block in the figure). All links going upwards from a block form an *upward LinkBlock*, and all links going downward towards a block form a *downward LinkBlock*. Flows are partitioned by both their source and destination blocks into *FlowBlocks*. This partitioning reduces concurrent updates, but does not eliminate them, as each upward LinkBlock is still updated by all FlowBlocks in the same source block. Similarly, downward LinkBlocks are updated by all FlowBlocks in the same destination block.

Eliminating concurrent updates. To eliminate concurrent updates completely, each FlowBlock works on private, local copies of its upward and downward LinkBlocks. The local copies are then *aggregated* into global copies. The algorithm then proceeds to update link prices on the global copies, and *distributes* the results back to FlowBlocks, so they again have local copies of the prices. Distribution follows the reverse of the aggregation pattern.

Figure 5 shows the LinkBlock aggregation pattern. Each aggregation step m combines LinkBlocks within each $2m \times 2m$ group of processes to the group diagonals, with the main diagonal aggregating upward LinkBlocks, and the secondary diagonal downward LinkBlocks. The aggregation scheme scales well with the number of cores. n^2 processors require only $\log_2 n$ steps rather than $\log_2 n^2$—the number of steps increases only every quadrupling of processors.

The aggregation pattern has uniform bandwidth requirements: when aggregating $2m \times 2m$ processors, each $m \times m$ sub-group sends and receives the same amount of LinkBlocks state to/from its neighbor sub-groups. Unlike FlowBlocks, whose size depends on the traffic pattern, each LinkBlock contains exactly the same number of links, making transfer latency more predictable.

Sending LinkBlocks is also much cheaper than sending FlowBlocks: datacenter measurements show average flow count per server at tens to hundreds of flows [18, 2], while LinkBlocks have a small constant number of links per server (usually between one and three).

Multi-machine allocator. The LinkBlock-FlowBlock partitioning distributes the allocator on multiple machines. Figure 5 shows a setup with four machines with 16 cores each. In steps (a) and (b), each machine aggregates LinkBlocks internally, then in (c), aggregation is performed across machines; each machine receives from one machine and sends to another. This arrangement scales to any $2^m \times 2^m$ collection of machines.

5 Evaluation

We evaluate Flowtune using a cluster deployment, microbenchmarks, and ns-2 and numeric simulations. ns-2 simulations allows comparison with state-of-the art schemes whose implementations are only readily avail-

able in the ns-2 simulator: pFabric [4], sfqCoDel [32], and XCP [26].

EC2 Experiments (§5.1)
(A) On Amazon EC2, Flowtune's sharing of available throughput is more fair than the baseline Linux implementation running Cubic.
(B) Flowtune makes transfers on EC2 more predictable: Many-to-One Coflow Completion Time was sped up by 1.61× in p95 and 1.24× in p90.

Multicore micro-benchmarks (§5.2)
(C) A multi-core implementation optimizes traffic from 384 servers on 4 cores in 8.29 µs. 64 cores schedule 4608 servers' traffic in 30.71 µs – around 2 network RTTs.

ns-2 Simulations (§5.3)
(D) Flowtune converges quickly to a fair allocation within 100 µs, orders of magnitude faster than other schemes.
(E) The amount of traffic to and from the allocator depends on the workload; it is $< 0.17\%$, 0.57%, and 1.13% of network capacity for the Hadoop, cache, and web workloads.
(F) Rate update traffic can be reduced by 69%, 64%, and 33% when allocating 0.95 of link capacities on the Hadoop, cache, and web workloads.
(G) As the network size increases, allocator traffic takes the same fraction of network capacity.
(H) Flowtune achieves low p99 flow completion time: 8.6×-10.9× and 1.7×-2.4× lower than DCTCP and pFabric on 1-packet flowlets, and 3.5×-3.8× than sfqCoDel on 10-100 packets.
(I) Flowtune keeps p99 network queuing delay under 8.9 µs, 12× less than DCTCP.
(J) Flowtune maintains a negligible rate of drops. sfqCoDel drops up to 8% of bytes, pFabric 6%.
(K) Flowtune achieves higher proportional-fairness score than DCTCP, pFabric, sfqCoDel, and XCP.

Numeric simulation (§5.4)
(L) Normalization is important; without it, NED overallocates links by up to 140 Gbits/s.
(M) F-NORM achieves over 99.7% of optimal throughput. U-NORM is not competitive.

5.1 Amazon EC2 deployment

We deployed Flowtune on 10 Amazon EC2 `c4.8xlarge` instances running Ubuntu 16.04 with 4.4.0 Linux kernels. One of the instances ran the allocator and had direct access to the NIC queues using SR-IOV. The other instances ran the workload.

Server module. We implemented the Flowtune client side using a kernel module, requiring no modification to applications. The module reports to the allocator when socket buffers transition between empty and non-empty, and enforces allocated rates by delaying packets when the rate limit is exceeded. An implementation could also change TCP slow-start and loss/marking behavior, but our implementation keeps those unchanged.

Protocol. Communication uses the Flowtune protocol over a variant of the Fastpass Control Protocol (FCP) for transport. The Flowtune protocol allows endpoints to process payloads without head-of-line blocking, so a dropped packet does not increase latency for non-dropped packets. The Flowtune protocol synchronizes state between the allocator and endpoints; when reacting to loss, instead of retransmitting old state, participants send the most recent state, and that only if the acknowledged state differs.

Allocator. The allocator is written in C++ and accesses NIC queues directly using the DPDK library. A hash table maps endpoints to their flow state, which the protocol maintains in synchronization with the endpoints. When allocated flow rates differ from the allocations acknowledged by the endpoints, the allocator triggers rate update messages.

Measurement. The experiment harness achieves accurate workload timing by measuring the clock offset of each instance using ntpdate. Before starting/stopping the workload, processes on the measured instances call nanosleep with appropriate amounts to compensate.

(A) Fairness. In an 8-to-1 experiment, eight senders start every 50 ms in sequence, and then finish similarly. Figure 6 shows the rates of each flow as the experiment progresses. Flowtune shares the throughput much more fairly than the baseline: the rates of the different flows overlap at equal division of throughput. The baseline rates oscillate, even with only 3 competing flows.

(B) Coflow completion time. Here, 8 senders each make 25 parallel transfers of 10 Mbytes to a single receiver. This transfer pattern models Spark aggregating data from worker cores, or a slice of a larger MapReduce shuffle stage. Figure 7 shows results from 100 runs with Flowtune vs. the baseline. Flowtune achieves more predictable results. The reduction in different percentiles are summarized in the following table.

Metric	Baseline	Flowtune	Speedup
median	1.859249	1.787622	1.04×
p90	2.341086	1.881433	1.24×
p95	3.050718	1.894544	1.61×

5.2 Multicore micro-benchmarks

We benchmarked NED's multi-core implementation on a machine with 8 Intel E7-8870 CPUs, each with 10 physi-

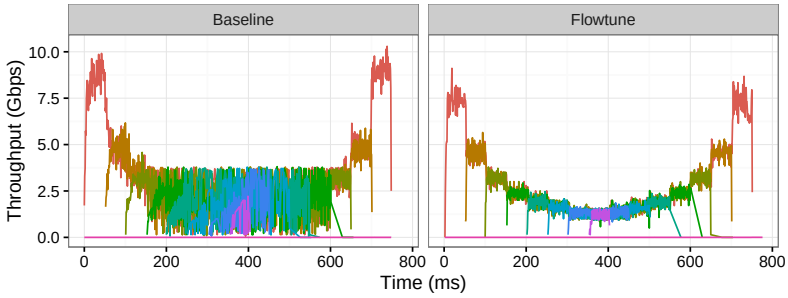


Figure 6: 8-to-1 experiment on Amazon EC2. Flowtune shares available throughput more fairly than baseline Linux.

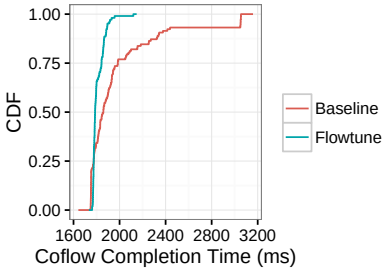


Figure 7: AWS EC2 data aggregation benchmarks. Flowtune coflows are more predictable and generally faster.

cal cores running at 2.4 GHz. We divided the network into 2, 4 and 8 blocks, giving runs with 4, 16, and 64 FlowBlocks. In the 4-core run, we mapped all FlowBlocks to the same CPU. With higher number of cores, we divided all FlowBlocks into groups of 2-by-2, and put two adjacent groups on each CPU.

(C) Iteration time. This micro-benchmark measures the average NED iteration time, i.e., the "Run 1 iteration" in Figure 1c. The following table shows the number of cycles taken for different choices of network sizes and loads:

Cores	Nodes	Flows	Cycles	Time
4	384	3072	19896.6	8.29 µs
16	768	6144	21267.8	8.86 µs
64	1536	12288	30317.6	12.63 µs
64	1536	24576	33576.2	13.99 µs
64	1536	49152	40628.5	16.93 µs
64	3072	49152	57035.9	23.76 µs
64	4608	49152	73703.2	30.71 µs

Rows 1-3 show run-times with increasing number of cores, rows 3-5 with increasing number of flows, and rows 5-7 with increasing number of endpoints. These results show general-purpose CPUs are able to optimize network allocations on hundred of nodes within microseconds.

Rate allocation for 49K flows from 4608 endpoints takes 30.71 µs, around 2 network RTTs, or 3 RTTs considering an RTT for control messages to obtain the rate. TCP takes tens of RTTs to converge – significantly slower.

Communication between CPUs in the aggregate and distribute steps took more than half of the runtime in all experiments, for example > 20 µs with 4068 nodes. This implies it should be straightforward to perform the aggregate and distribute steps on multiple servers in a cluster using commodity hardware and kernel-bypass libraries.

Note that this benchmark only captures the computation required to optimize flows; communication between the servers and the allocator is evaluated in §5.3.2, and discussed in §6.

Throughput scaling and comparison to Fastpass. Flowtune scales to larger networks than Fastpass, which reported 2.2 Tbit/s on 8 cores. Fastpass performs per-packet work, so its scalability declines with increases in link speed. Flowtune schedules flowlets, so allocated rates scale proportionally with the network links. The benchmark results above show that on 40 Gbits/s links, 4 cores allocate 15.36 Tbit/s, and 64 cores allocate 184 Tbit/s on 64 cores in under 31 µs, 10.4× more throughput per core on 8× more cores – an 83× throughput increase over Fastpass.

5.3 ns-2 simulations

Model. All control traffic shares the network with data traffic and experiences queuing and packet drops. Control payloads are transmitted using TCP, and are only processed after all payload bytes arrive at their destinations.

Topology. The topology is a two-tier full-bisection topology with 4 spine switches connected to 9 racks of 16 servers each, where servers are connected with a 10 Gbits/s link. It is the same topology used in [4]. Links and servers have 1.5 and 2 microsecond delays respectively, for a total of 14 µs 2-hop RTT and 22 µs 4-hop RTT, commensurate with measurements we conducted in a large datacenter.

Workload. To model micro-bursts, flowlets follow a Poisson arrival process. Flowlet size distributions are according to the Web, Cache, and Hadoop workloads published by Facebook [34]. Appendix C has more information on the CDFs used. The Poisson rate at which flows enter the system is chosen to reach a specific average server load, where 100% load is when the rate equals server link capacity divided by the mean flow size. Unless otherwise specified, experiments use the Web workload, which has the highest rate of changes and hence stresses Flowtune the most among the three workloads. Sources and destinations are chosen uniformly at random.

Servers. When opening a new connection, servers start a regular TCP connection, and in parallel send a notification

to the allocator. Whenever a server receives a rate update for a flow from the allocator, it opens the flow's TCP window and paces packets on that flow according to the allocated rate.

Flowtune allocator. The allocator performs an iteration every 10 µs. We found that for NED parameter γ in the range $[0.2, 1.5]$, the network exhibits similar performance; experiments have $\gamma = 0.4$.

Flowtune control connections. The allocator is connected using a 40 Gbits/s link to each of the spine switches. Allocator–server communication uses TCP with a 20 µs minRTO and 30 µs maxRTO. Notifications of flowlet start, end, and rate updates are encoded in 16, 4, and 6 bytes plus the standard TCP/IP overheads. Updates to the allocator and servers are only applied when the corresponding bytes arrive, as in ns2's TcpApp.

5.3.1 Fast convergence

To show how fast the different schemes converge to a fair allocation, we ran five senders and one receiver. Starting with an empty network, every 10 ms one of the senders would start a flow to the receiver. Thereafter, every 10 ms one of the senders stops.

(D) Convergence comparison. Figure 8 shows the rates of each of the flows as a function of time. Throughput is computed at 100 µs intervals; smaller intervals make very noisy results for most schemes. Flowtune achieves an ideal sharing between flows: N flows each get $1/N$ of bandwidth. This changes happens within one averaging interval (100 µs). DCTCP takes several milliseconds to approach the fair allocation, and even then traffic allocations fluctuate. pFabric doesn't share fairly; it prioritizes the flow with least remaining bytes and starves the other flows. sfqCoDel reaches a fair allocation quickly, but packet drops cause the application-observed throughput to be extremely bursty: the application sometime receives nothing for a while, then a large amount of data when holes in the window are successfully received. XCP is slow to allocate bandwidth, which results in low throughputs during most of the experiment.

5.3.2 Rate-update traffic

Flowtune only changes allocations on flowlet start and stop events, so when these events are relatively infrequent, the allocator could send relatively few updates every second. On the other hand, since the allocator optimizes utility across the entire network, a change to a single flow could potentially change the rates of all flows in the network. This section explores how much traffic is generated to and from the allocator.

The allocator notifies servers when the rates assigned to flows change by a factor larger than a threshold. For example, with a threshold of 0.01, a flow allocated 1 Gbit/s will only be notified when its rate changes to above 1.01 or below 0.99 Gbits/s. To make sure links are not over-utilized, the allocator adjusts the available link capacities by the threshold; with a 0.01 threshold, the allocator would allocate 99% of link capacities.

(E) Amount of update traffic. Figure 9 shows the amount of traffic sent to and from the allocator as a fraction of total network capacity, with a notification threshold of 0.01. The Web workload, which has the smallest mean flow size, also incurs the most update traffic: 1.13% of network capacity. At 0.8 load, the network will be 80% utilized, with 20% unused, so update traffic is well below the available headroom. Hadoop and Cache workloads need even less update traffic: 0.17% and 0.57%. Scaling the rate updates to large networks is discussed in §6.

Traffic from servers to the allocator is substantially lower than from the allocator to servers: servers only communicate flowlet arrival and departures, while the allocator can potentially send many updates per flowlet.

(F) Reducing update traffic. Increasing the update threshold reduces the volume of update traffic and the processing required at servers. Figure 10 shows the measured reduction in update traffic for different thresholds compared to the 0.01 threshold in Figure 9. Notifying servers of changes of 0.05 or more of previous allocations saves up to 69%, 64% and 33% of update traffic for the Hadoop, Cache, and Web workloads.

(G) Effect of network size on update traffic. An addition or removal of a flow in one part of the network potentially changes allocations on the entire network. As the network grows, does update traffic also grow, or are updates contained? Figure 11 shows that as the network grows from 128 servers up to 2048 servers, update traffic takes the same fraction of network capacity — there is no debilitating cascading of updates that increases update traffic. This result shows that the threshold is effective at limiting the cascading of updates to the entire network.

5.3.3 Comparison to prior schemes

We compare Flowtune to DCTCP [2], pFabric [4], XCP [26], and Cubic+sfqCoDel [32].

(H) 99th percentile FCT. For datacenters to provide faster, more predictable service, tail latencies must be controlled. Further, when a user request must gather results from tens or hundreds of servers, p99 server latency quickly dominates user experience [11].

Figure 12 shows the improvement in 99th percentile flow completion time achieved by switching from different schemes to Flowtune. To summarize flows of different lengths to the different size ranges ("1-10 packets", etc.), we normalize each flow's completion time by the time

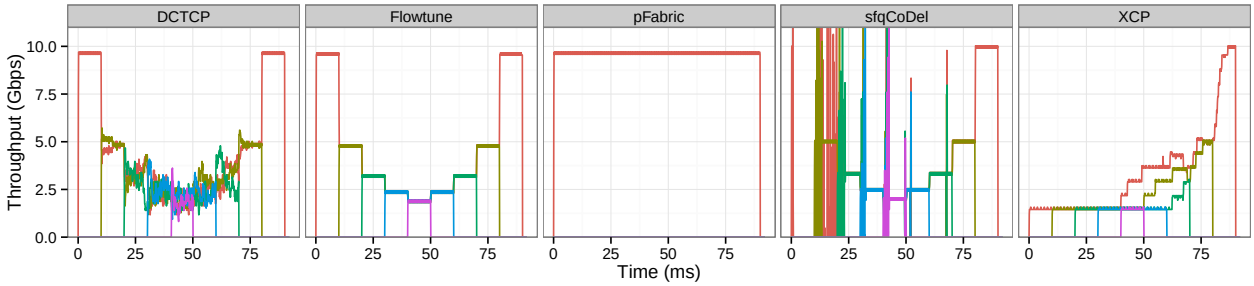


Figure 8: Flowtune achieves a fair allocation within 100 µs of a new flow arriving or leaving. In the benchmark, every 10 ms a new flow is added up to 5 flows, then flows finish one by one. DCTCP approaches a fair allocation after several milliseconds. pFabric, as designed, doesn't share the network among flows. sfqCoDel gets a fair allocation quickly, but retransmissions cause the application to observe bursty rates. XCP is conservative in handing out bandwidth and so converges slowly.

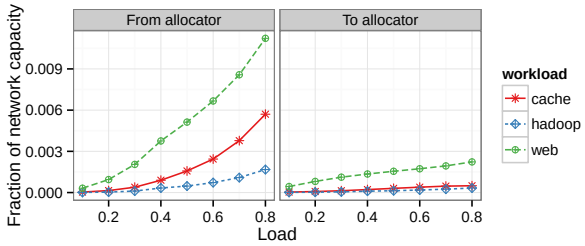


Figure 9: Overhead with Hadoop, cache, and Web workloads is $<$ 0.17%, 0.57%, and 1.13% of network capacity.

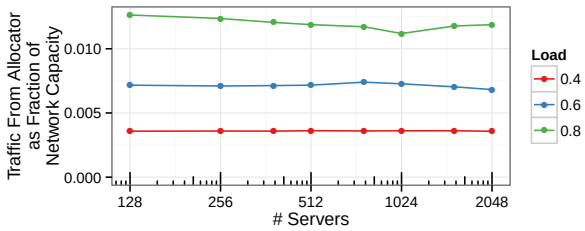


Figure 11: The fraction of rate-update traffic remains constant as the network grows from 128 to 2048 servers.

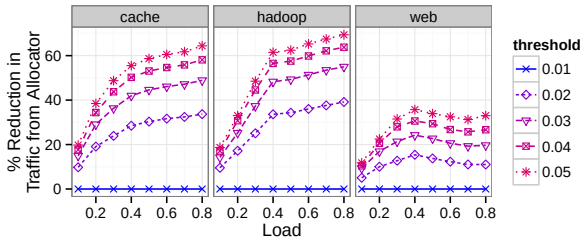


Figure 10: Notifying servers when rates change by more than a threshold substantially cuts control traffic volume.

it would take to send out and receive all its bytes on an empty network.

Flowtune preforms better than DCTCP on short flows: 8.6×-10.9× lower p99 FCT on 1-packet flows and 2.1×-2.9× on 1-10 packet flows. This happens because DCTCP has high p99 queuing delay, as shown in the next experiment.

Overall, pFabric and Flowtune have comparable performance, with Flowtune better on some flow sizes, pFabric on others. Note, however, that Flowtune achieves this performance without requiring any changes to networking hardware. Flowtune achieves 1.7×-2.4× lower p99 FCT on 1-packet flows, and up to 2.4× on large flows. pFabric performs well on flows 1-100 packets long, with similar ratios. pFabric is designed to prioritize short flows, which explains its performance.

sfqCoDel has comparable performance on large flows, but is 3.5×-3.8× slower on 10-100 packets at high load and 2.1×-2.4× slower on 100-1000 packet flows at low load. This is due to sfqCoDel's high packet loss rate. Cubic handles most drops using SACKs, except at the end of the flow, where drops cause timeouts. These timeouts are most apparent in the medium-sized flows. XCP is conservative in allocating bandwidth (§5.3.1), which causes flows to finish slowly.

(I) Queuing delay. The following experiments collected queue lengths, drops, and throughput from each queue every 1 ms. Figure 13 shows the 99th percentile queuing delay on network paths, obtained by examining queue lengths. This queuing delay has a major contribution to 1-packet and 1-10 packet flows. Flowtune has near-empty queues, whereas DCTCP's queues are 12× longer, contributing to the significant speedup shown in Figure 12. XCP's conservative allocation causes its queues to remain shorter. pFabric and sfqCoDel maintain relatively long queues, but the comparison is not apples-to-apples because packets do not traverse their queues in FIFO order.

(J) Packet drops. Figure 14 shows the rate at which the network drops data, in Gigabits per second. At 0.8 load,

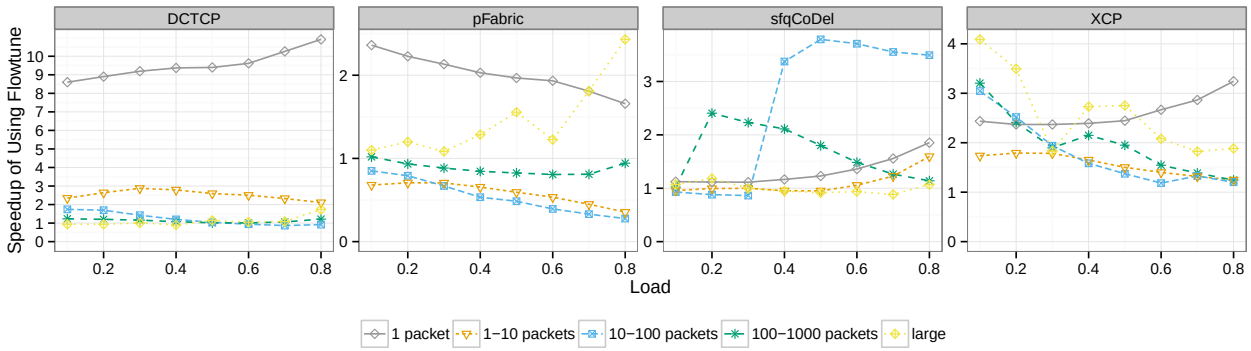


Figure 12: Improvement in 99th percentile flow completion time with Flowtune. Note the different scales of the y axis.

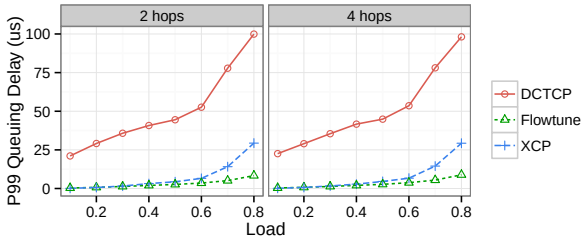


Figure 13: Flowtune keeps 2-hop and 4-hop 99th-percentile queuing delays below 8.9 µs. At 0.8 load, XCP has 3.5× longer queues, DCTCP 12×. pFabric and sfqCoDel do not maintain FIFO ordering so their p99 queuing delay could not be inferred from sampled queue lengths.

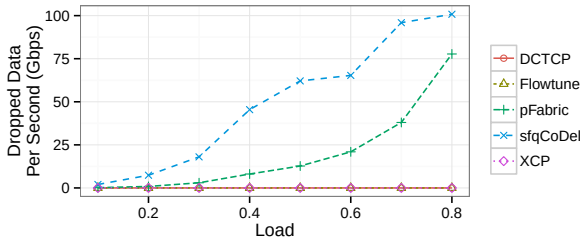


Figure 14: pFabric and sfqCoDel have a significant drop rate (1-in-13 for sfqCoDel). Flowtune, DCTCP, and XCP drop negligible amounts.

sfqCoDel servers transmit at 1279 Gbits/s (not shown), and the network drops over 100 Gbits/s, close to 8%. These drops in themselves are not harmful, but timeouts due to these drops could result in high p99 FCT, which affects medium-sized flows (figure 12). Further, in a datacenter deployment of sfqCoDel, servers would spend many CPU cycles in slow-path retransmission code. pFabric's high drop rate would also make it prone to higher server CPU usage, but its probing and retransmission schemes mitigate high p99 FCT. Flowtune, DCTCP, and XCP drop negligible amounts.

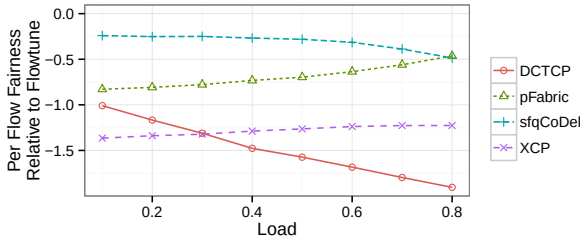


Figure 15: Comparison of proportional fairness of different schemes, i.e., $\sum \log_2(\text{rate})$. Flowtune allocates flows closer to their proportional-fair share.

(K) Fairness. Figure 15 shows the proportional-fairness per-flow score of the different schemes normalized to Flowtune's score. A network where flows are assigned rates r_i gets score $\sum_i \log_2(r_i)$. This translates to gaining a point when a flow gets 2× higher rate, losing a point when a flow gets 2× lower rate. Flowtune has better fairness than the compared schemes: a flow's fairness score has on average 1.0-1.9 points more in Flowtune than DCTCP, 0.45-0.83 than pFabric, 1.3 than XCP, and 0.25 than CoDel.

5.4 Numerical simulations

Experiments in this section compared different NUM optimizers using numerical simulations. Simulations ran the web flow size distribution described in §5.3.

(L) Over-allocation in NUM. Figure 16 shows the total amount of over-capacity allocations when there is no normalization. FGM is the Fast Weighted Gradient Method [7]. The -RT variants are optimized implementations which use single-point floating point operations and some numeric approximations for speed. NED over-allocates more than Gradient because it is more aggressive at adjusting prices when flowlets arrive and leave. FGM does not handle the stream of updates well, and its allocations become unrealistic at even moderate loads.

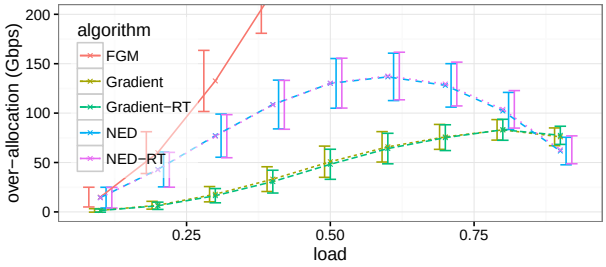


Figure 16: Normalization is necessary; without it, optimization algorithms allocate more than link capacities.

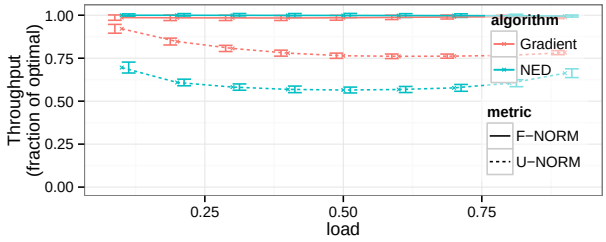


Figure 17: Normalizing with F-NORM achieves close to optimal throughput, while avoiding over-capacity allocations. U-NORM's throughput is low in comparison.

(M) Normalizer performance. We ran Gradient and NED on the same workload and recorded their throughput. After each iteration, we ran a separate instance of NED until it converged to the optimal allocation. Figure 17 shows U-NORM and F-NORM throughputs as a fraction of the optimal. F-NORM scales each flow based on the over-capacity allocations of links it traverses, achieving over 99.7% of optimal throughput with NED (98.4% with Gradient). In contrast, U-NORM scales flow throughput too aggressively, hurting overall performance. Gradient suffers less from U-NORM's scaling, because it adjusts rates slowly and does not over-allocate as much as NED. Note that NED with F-NORM allocations occasionally slightly exceed the optimal allocation, but not the link capacities. Rather, the allocation gets more throughput than the optimal at the cost of being a little unfair to some flows.

6 Discussion

Fault-tolerance: In Flowtune, the allocated rates have a temporary lifespan, and new allocated rates must arrive every few tens of microseconds. If the allocator fails, the rates expire and endpoint congestion control (e.g., TCP) takes over, using the previously allocated rates as a starting point.

This is a more attractive plan than in Fastpass. When the Fastpass arbiter fails, the network has no idea who should transmit next. Falling back to TCP requires the endpoints to go through slow-start before finding a good allocation. In Flowtune, the network continues to operate with close-to-optimal rates during allocator fail-over.

Path discovery: The allocator knows each flow's path through the network. Routing information can be computed from the network state: in ECMP-based networks, given the ECMP hash function and switch failure notifications; in SDN-based networks, given controller decisions; and in MPLS-based [14] networks, given the MPLS configuration stream. In VL2 [18]-like networks where endpoints tunnel packets to a core switch for forwarding to the destination, and in static-routed network where endpoints have multiple subnets for different paths and the choice of subnet dictates a packet's path, endpoints can send chosen routes to the allocator.

Using Flowtune with TCP: In some settings, it is advantageous to use Flowtune to rate-limit traffic, while still using TCP. One such setting is when path information is not available. Flowtune can model the network as the servers connected to one big switch. Then, Flowtune would manages rates on the edge links between servers and the top-of-rack switches, while TCP congestion control would handle contention in the core of the network.

Flow startup; mice and elephants: When using Flowtune with TCP, the system can allow servers to start transmitting their flowlets before an allocation has arrived; the allocator would reserve a small fraction of link capacity to accomodate these flows. This allows short mice flows to finish quickly, without paying for the RTT to the allocator, and simplifies fault tolerance: upon allocator failure, the endpoints automatically use TCP on new flowlets without incurring timeouts.

Handling network failure: When links and switches fail, the network re-routes flows via alternate paths. For example, ECMP would hash flows onto the set of available links. Re-routing flows without notifying the allocator can cause queuing and packet loss on their new paths, since the allocator computes flow rates according to their old paths.

When timely notifications of re-routing events are not possible, the system can run traffic over TCP as discussed above. While re-routing information is not available, TCP gracefully handles over-allocations. An alternative is detecting failure using an external system like Pingmesh [19], and then triggering path re-discovery or re-routing for affected flows. While correct path information is being obtained, the allocator can mitigate link over-allocation by zeroing the capacity of failed links.

External traffic: Most datacenters do not run in isolation; they communicate with other datacenters and users on the Internet. A Flowtune cluster must be able to accept flows that are not scheduled by the allocator. As in Fastpass,

Flowtune could prioritize or separately schedule external traffic, or adopt a different approach. With NED, it is straightforward to dynamically adjust link capacities or add dummy flows for external traffic; a "closed loop" version of the allocator would gather network feedback observed by endpoints, and adjust its operation based on this feedback. The challenge here is what feedback to gather, and how to react to it in a way that provides some guarantees on the external traffic performance.

Scaling to larger networks: Although the allocator scales to multiple servers, the current implementation is limited to two-tier topologies. Beyond a few thousand endpoints, some networks add a third tier of spine switches to their topology that connects two-tier *pods*. Assigning a full pod to one block would create huge blocks, limiting allocator parallelism. On the other hand, the links going into and out of a pod are used by all servers in a pod, so splitting a pod to multiple blocks creates expensive updates. An open question is whether the FlowBlock/LinkBlock abstraction can generalize to 3-tier Clos networks, or if a new method is needed.

Another approach to scaling would be running a seperate Flowtune allocator per pod, each controller treating incoming inter-pod traffic as external traffic (as discussed above). This would allow each pod to optimize its objective function on its egress inter-pod flows, but the network will not be able to globally optimize inter-pod traffic.

More scalable rate update schemes: Experiments in §5.3.2 show that rate updates have a throughput overhead of 1.12% at 0.8 load, so each allocator NIC can update 89 servers. Note that 0.8 load is on the extreme high end in some datacenters: one study reports 99% of links are less than 10% loaded, and heavily-loaded links utilize roughly 5× the lightly loaded ones [34]. Lower load translates directly to reduced update traffic (Figure 9).

In small deployments of a few hundred endpoints, it might be feasible to install a few NICs in the allocator. Figure 10 shows how increasing the update threshold reduces update traffic, which can help scale a little further, but as deployments grow to thousands of endpoints, even the reduced updates can overwhelm allocator NICs.

Sending tiny rate updates of a few bytes has huge overhead: Ethernet has 64-byte minimum frames and preamble and interframe gaps, which cost 84-bytes, even if only eight byte rate updates are sent. A straightforward solution to scale the allocator 10× would be to employ a group of intermediary servers that handle communication to a subset of individual endpoints. The allocator would send an MTU to each intermediary with all updates to the intermediary's endpoints. The intermediary would in turn forward rate updates to each endpoint.

Hypervisors: A Flowtune endpoint needs to send flowlet start/stop notifications to the allocator, and rate-limit flows based on received allocations. A hypervisor can accomplish this without VM support by interposing itself on VM network I/O (e.g., using a vSwitch), maintaining per-flow queues, and scheduling outgoing packets. However, this approach precludes direct VM access to NIC queues (e.g., using SR-IOV) and its associated performance advantages. A potential direction could be adding hardware support for flow notification and pacing to NICs.

Detecting flowlets: Detection of when flowlets start and end can be done in the operating system, in the hypervisor (as discussed above), in a network appliance/switch (similar to a hypervisor implementation), or in some implementations the applications could participate in Flowtune directly. With OS and application flowlet detection, flowlets are clearly delineated by different `send()` socket calls, and timers are not required to detect a flowlet's end. This accurate detection is more economical to the system, since a rate is not allocated in vain to an empty flowlet while waiting for its timer to expire. Moreover, knowing the exact flowlet size allows the OS and applications to provide the allocator with advance notification of flowlet endings, further reducing wasted allocations.

When a new `send()` socket call arrives in the middle of a flowlet, an implementation can choose to coalesce the new data into the existing flowlet, and notify the allocator only when all data has finished. This is beneficial when the utility function is the same for both socket calls: the allocator will output the same rate if there are two back-to-back flowlets or one large flowlet, and coalescing helps reduce communication overhead.

7 Related work

Rate allocation. NUMFabric [31] also uses NUM to assign network rates, however switches must be modified to support its xWI protocol. Unlike Flowtune, it is distributed, so an iteration time is coupled with network RTT and the system cannot apply global normalization to make all traffic admissible.

Several systems control datacenter routes and rates, but are geared for inter-datacenter traffic. BwE [28] groups flows hierarchically and assigns a max-min fair allocation at each level of the hierarchy every 5-10 seconds on WAN links (similar time-scale to B4 [25]), and SWAN [23] receives demands from non-interactive services, computes rates, and reconfigures OpenFlow switches every 5 minutes. Flowtune supports a richer set of utility functions, with orders of magnitude smaller update times.

Hedera [1] gathers switch statistics to find elephant flows and reroutes those to avoid network hotspots. It is complementary to Flowtune: integrating the two systems can give Hedera its required information with very low latency. Mordia [15] and Datacenter TDMA [40] compute matchings between sources and destinations using

gathered statistics, and at any given time, only flows of a single matching can send. While matchings are changed relatively frequently, the set of matchings is updated infrequently (seconds). In contrast, Flowtune updates allocations within tens of microseconds.

NED. The first-order methods [27, 29, 37] do not estimate $H_{\ell\ell}$ or use crude proxies. Gradient projection [29] adjusts prices with no weighting. Fast Weighted Gradient [7] uses a crude upper bound on the convexity of the utility function as a proxy for $H_{\ell\ell}$.

The Newton-like method [6], like NED, strives to use $H_{\ell\ell}$ to normalize price updates, but it uses network measurements to estimate its value. These measurements increase convergence time and have associated error; we have found the algorithm is unstable in several settings. Flowtune, in contrast, computes $H_{\ell\ell}$ explicitly from flow utilities, saving the time required to obtain estimates, and getting an error-free result. Appendix B discusses the Gradient, Newton and Newton-like methods in more detail.

Recent work [41] has a different formulation of the problem, with equality constraints rather than inequalities. While the scheme holds promise for faster convergence, iterations are much more involved and hence slower to compute, making the improvement questionable. Accelerated Dual Descent [43] does not use the flow model: it doesn't care what destination data arrives at, only that all data arrives at *some* destination. However, the method is notable for updating a link's price p_ℓ based not only on the link's current and desired throughput, but also on how price changes to other links p_k affect it. Adapting the method to the flow setting could reduce the number of required iterations to convergence (again at the cost of perhaps increasing iteration runtime).

8 Conclusion

This paper made the case for *flowlet control* for datacenter networks. We developed Flowtune using this idea and demonstrated that it converges to an optimal allocation of rates within a few packet-times, rather than several RTTs. Our experiments show that Flowtune outperforms DCTCP, pFabric, Cubic-over-sfqCoDel, and XCP in various datacenter settings; for example, it achieves 8.6×-10.9× and 2.1×-2.9× lower p99 FCT for 1-packet and 1-10 packet flows compared to DCTCP.

Compared to Fastpass, Flowtune scales to 8× more cores and achieves 10.4× higher throughput per core, does not require allocator replication for fault-tolerance, and achieves weighted proportional-fair rate allocations quickly in between 8.29 μs and 30.71 μs ($\leq$2 RTTs) for networks that have between 384 and 4608 nodes.

Acknowledgements

We thank Omar Baldonado, Chuck Thacker, Prabhakaran Ganesan, Songqiao Su, Kirtesh Patil, Petr Lapukhov, Neda Beheshti, Mohana Prasad, Mohammad Alizadeh, James Zeng, Sandeep Hebbani, Jasmeet Bagga, Dinesh Bharadia, Chris Davies, and Doug Weimer for helpful discussions. We are grateful for Facebook's support of Perry through a Facebook Fellowship. Balakrishnan was supported in part by NSF grants 1526791 and 1407470, and Shah by NSF grant 1523546. We thank the industrial members of the MIT Center for Wireless Networks and Mobile Computing for their support and encouragement.

A Why price duality works

The utility function U_s for each $s \in S$ is a strictly concave function and hence the overall objective $\sum_s U_s$ in (2) is strictly concave. The constraints in (2) are linear. The capacity of each link is strictly positive and finite. Each flow passes through at least one link, i.e. $L(s) \neq \emptyset$ for each $s \in S$. Therefore, the set of feasible solutions for (2) is non-empty, bounded and convex. The Lagrangian of (2) is

$$\mathscr{L}(\mathbf{x},\mathbf{p}) = \sum_{s\in S} U_s(x_s) - \sum_{\ell\in L} p_\ell \Big(\sum_{s\in S(\ell)} x_s - c_\ell \Big). \quad (7)$$

with dual variables p_ℓ, and the dual function is defined as

$$D(\mathbf{p}) = \max\ \mathscr{L}(\mathbf{x},\mathbf{p}) \text{ over } x_s \geq 0, \text{ for all } s \in S. \quad (8)$$

The dual optimization problem is given by

$$\min\ D(\mathbf{p}) \text{ over } p_\ell \geq 0, \text{ for all } \ell \in L. \quad (9)$$

From Slater's condition in classical optimization theory, the utility of the solution of (2) is equal to its Lagrangian dual's (9), and given the optimal solution $\mathbf{p}^\star$ of (9) it is possible to find the optimal solution for (2) from (8), i.e., using the rate update step. More details on solving NUM using Lagrange multipliers appear in [27, 6].

B Related NUM algorithms

This appendix surveys three related NUM algorithms.

Gradient. Arguably the simplest algorithm for adjusting prices is Gradient projection [29], which adjusts prices directly from the amount of over-allocation:

$$p_\ell \leftarrow p_\ell + \gamma G_\ell.$$

Gradient's shortcoming is that it doesn't know how sensitive flows are to a price change, so it must update prices very gently (i.e., γ must be small). This is because depending on flow utility functions, large price updates

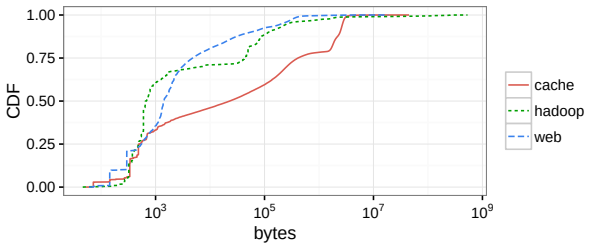


Figure 18: CDF of flow size distributions (reproduced from [34]) used in §5.

Metric	cache	hadoop	web
mean	567658.4	1296082.4	33105.3
median	22924	651	1419
p90	2432831	117471	55179
p95	2716140	266706	208966
p99	3131038	6405830	417147
p999	5663439	251359175	2560769

Table 1: Statistics of the different flow size distributions (bytes).

might cause flows to react very strongly and change rates dramatically, causing oscillations in rates and failure to converge. This results in very timid price updates that make Gradient slow to converge.

Newton's method. Unlike the gradient method, Newton's method takes into account second-order effects of price updates. It adjusts the price on link ℓ based not only on how flows on ℓ will react, but also based on how price changes to all other links impact flows on ℓ:

$$\mathbf{p} \leftarrow \mathbf{p} - \gamma \mathbf{G} H^{-1},$$

where H is the Hessian matrix. This holistic price update makes Newton's method converge quickly, but also makes computing new prices expensive: inverting the Hessian on CPUs is impractical within Flowtune's time constraints.

The Newton-like method. An approximation to the Newton method was proposed in [6]. The Newton-like method estimates how sensitive flows are to price changes, by observing how price changes impact network throughput. Prices are then updated accordingly: inversely proportional to the estimate of price-sensitivity. The drawback is that network throughput must be averaged over relatively large time intervals, so estimating the diagonal is slow.

C Simulation CDFs

This section reproduces the flow size distribution graphs from [34], for completeness. Data from the paper has been open-sourced in the "Facebook Network Analytics Data Sharing" Facebook group. The distributions are based on the "all" category from the original publication.

Figure 18 shows the flow size CDF. Table 1 summarizes statistics of the different workloads.

References

[1] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI* (2010).

[2] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data Center TCP (DCTCP). In *SIGCOMM* (2010).

[3] Alizadeh, M., Kabbani, A., Edsall, T., Prabhakar, B., Vahdat, A., and Yasuda, M. Less is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI* (2012).

[4] Alizadeh, M., Yang, S., Sharif, M., Katti, S., McKeown, N., Prabhakar, B., and Shenker, S. pFabric: Minimal near-optimal datacenter transport. In *SIGCOMM* (2013).

[5] Andreyev, A. Introducing data center fabric, the next-generation Facebook data center network.

[6] Athuraliya, S., and Low, S. H. Optimization Flow Control with Newton-like Algorithm. *Telecommunication Systems 15*, 3-4 (2000), 345–358.

[7] Beck, A., Nedic, A., Ozdaglar, A., and Teboulle, M. A Gradient Method for Network Resource Allocation Problems. *IEEE Trans. on Control of Network Systems 1*, 1 (2014), 64–73.

[8] Brakmo, L. S., O'Malley, S. W., and Peterson, L. L. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *SIGCOMM* (1994).

[9] Chen, Y., Alspaugh, S., and Katz, R. H. Design insights for mapreduce from diverse production workloads. In *Tech. Rep. EECS-2012-17* (2012), UC Berkeley.

[10] Chowdhury, M., Zhong, Y., and Stoica, I. Efficient coflow scheduling with varys. In *SIGCOMM* (2014).

[11] Dean, J., and Barroso, L. A. The tail at scale. *Comm. of the ACM* (2013).

[12] Demers, A., Keshav, S., and Shenker, S. Analysis and Simulations of a Fair-Queueing Algorithm. *Internetworking: Research and Experience V*, 17 (1990), 3–26.

[13] Dong, M., Li, Q., Zarchy, D., Godfrey, P. B., and Schapira, M. PCC: Re-architecting Congestion Control for Consistent High Performance. In *NSDI* (2015).

[14] Elwalid, A., Jin, C., Low, S., and Widjaja, I. MATE: MPLS Adaptive Traffic Engineering. In *INFOCOM* (2001).

[15] Farrington, N., Porter, G., Fainman, Y., Papen, G., and Vahdat, A. Hunting Mice with Microsecond Circuit Switches. In *HotNets* (2012).

[16] Floyd, S. TCP and Explicit Congestion Notification. *CCR 24*, 5 (Oct. 1994).

[17] Floyd, S., and Jacobson, V. Random Early Detection Gateways for Congestion Avoidance. *IEEE ACM Trans. on Net. 1*, 4 (Aug. 1993).

[18] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM* (2009).

[19] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., ET AL. Pingmesh: A large-scale system for data center network latency measurement and analysis. *SIGCOMM* (2015).

[20] HA, S., RHEE, I., AND XU, L. CUBIC: A new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review 42*, 5 (2008), 64–74.

[21] HOE, J. C. Improving the start-up behavior of a congestion control scheme for tcp. In *SIGCOMM* (1996).

[22] HONG, C. Y., CAESAR, M., AND GODFREY, P. Finishing Flows Quickly with Preemptive Scheduling. In *SIGCOMM* (2012).

[23] HONG, C.-Y., KANDULA, S., MAHAJAN, R., ZHANG, M., GILL, V., NANDURI, M., AND WATTENHOFER, R. Achieving High Utilization with Software-Driven WAN. In *SIGCOMM* (2013).

[24] JACOBSON, V. Congestion Avoidance and Control. In *SIGCOMM* (1988).

[25] JAIN, S., KUMAR, A., MANDAL, S., ONG, J., POUTIEVSKI, L., SINGH, A., VENKATA, S., WANDERER, J., ZHOU, J., ZHU, M., ZOLLA, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. B4: Experience with a Globally-deployed Software Defined Wan. In *SIGCOMM* (2013).

[26] KATABI, D., HANDLEY, M., AND ROHRS, C. Congestion Control for High Bandwidth-Delay Product Networks. In *SIGCOMM* (2002).

[27] KELLY, F. P., MAULLOO, A. K., AND TAN, D. K. Rate Control for Communication Networks: Shadow prices, Proportional Fairness and Stability. *Journal of the Operational Research Society* (1998), 237–252.

[28] KUMAR, A., JAIN, S., NAIK, U., RAGHURAMAN, A., KASINADHUNI, N., ZERMENO, E. C., GUNN, C. S., AI, J., CARLIN, B., AMARANDEI-STAVILA, M., ROBIN, M., SIGANPORIA, A., STUART, S., AND VAHDAT, A. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *SIGCOMM* (2015).

[29] LOW, S. H., AND LAPSLEY, D. E. Optimization Flow Control–I: Basic Algorithm and Convergence. *IEEE/ACM Trans. on Networking 7*, 6 (1999), 861–874.

[30] MITTAL, R., DUKKIPATI, N., BLEM, E., WASSEL, H., GHOBADI, M., VAHDAT, A., WANG, Y., WETHERALL, D., ZATS, D., ET AL. TIMELY: RTT-based congestion control for the datacenter. In *SIGCOMM* (2015).

[31] NAGARAJ, K., BHARADIA, D., MAO, H., CHINCHALI, S., ALIZADEH, M., AND KATTI, S. NUMFabric: Fast and Flexible Bandwidth Allocation in Datacenters. In *SIGCOMM* (2016).

[32] NICHOLS, K., AND JACOBSON, V. Controlling queue delay. *Communications of the ACM 55*, 7 (2012), 42–50.

[33] PERRY, J., OUSTERHOUT, A., BALAKRISHNAN, H., SHAH, D., AND FUGAL, H. Fastpass: A centralized "zero-queue" datacenter network. In *SIGCOMM* (2014).

[34] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the social network's (datacenter) network. In *SIGCOMM* (2015).

[35] SHREEDHAR, M., AND VARGHESE, G. Efficient Fair Queueing Using Deficit Round Robin. In *SIGCOMM* (1995).

[36] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., KANAGALA, A., PROVOST, J., SIMMONS, J., TANDA, E., WANDERER, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *SIGCOMM* (2015).

[37] SRIKANT, R. *The mathematics of Internet congestion control*. Springer Science & Business Media, 2012.

[38] TAI, C., ZHU, J., AND DUKKIPATI, N. Making Large Scale Deployment of RCP Practical for Real Networks. In *INFOCOM* (2008).

[39] TAN, K., SONG, J., ZHANG, Q., AND SRIDHARAN, M. A compound TCP approach for high-speed and long distance networks. In *INFOCOM* (2006).

[40] VATTIKONDA, B. C., PORTER, G., VAHDAT, A., AND SNOEREN, A. C. Practical TDMA for Datacenter Ethernet. In *EuroSys* (2012).

[41] WEI, E., OZDAGLAR, A., AND JADBABAIE, A. A Distributed Newton Method for Network Utility Maximization–I: Algorithm. *IEEE Trans. on Automatic Control 58*, 9 (2013), 2162–2175.

[42] WINSTEIN, K., SIVARAMAN, A., AND BALAKRISHNAN, H. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *NSDI* (2013).

[43] ZARGHAM, M., RIBEIRO, A., OZDAGLAR, A., AND JADBABAIE, A. Accelerated dual descent for network flow optimization. *IEEE Trans. on Automatic Control 59*, 4 (2014), 905–920.

Flexplane: An Experimentation Platform for Resource Management in Datacenters

Amy Ousterhout, Jonathan Perry, Hari Balakrishnan (MIT CSAIL), Petr Lapukhov (Facebook)

Abstract

Flexplane enables users to program data plane algorithms and conduct experiments that run real application traffic over them at hardware line rates. Flexplane explores an intermediate point in the design space between past work on software routers and emerging work on programmable hardware chipsets. Like software routers, Flexplane enables users to express resource management schemes in a high-level language (C++), but unlike software routers, Flexplane runs at close to hardware line rates. To achieve these two goals, a centralized emulator faithfully *emulates*, in real-time on a multi-core machine, the desired data plane algorithms with very succinct representations of the original packets. Real packets traverse the network when notified by the emulator, sharing the same fate and relative delays as their emulated counterparts.

Flexplane accurately predicts the behavior of several network schemes such as RED and DCTCP, sustains aggregate throughput of up to 760 Gbits/s on a 10-core machine ($\approx 20\times$ faster than software routers), and enables experiments with real-world operating systems and applications (e.g., Spark) running on diverse network schemes at line rate, including those such as HULL and pFabric that are not available in hardware today.

1 Introduction

Recently the networking community has witnessed a renewed flurry of activity in the area of programmability, with the goal of enabling experimentation and innovation [17, 25, 34, 38]. Programmable networks facilitate experimentation at several different levels: researchers can experiment with new network protocols, network operators can test out new protocols before deployment, developers can debug applications with new network optimizations, and students can implement, run, and measure network algorithms on real hardware.

Much of this work has focused on the control plane; platforms for programming resource management algorithms in the data plane remain limited. Examples of such algorithms include:

- *Queue management*: what packet drop or ECN-marking [23] policy should be used? Examples include RED [24], DCTCP [10], HULL [11], and D^{2}TCP [50].
- *Scheduling*: which queue's packet should be sent next on a link? Examples include weighted fair queueing (WFQ) [20], stochastic FQ [37], priority queueing, deficit round-robin (DRR) [44], hierarchical FQ [16], pFabric [12], and LSTF [39].
- *Explicit control*: how should dynamic state in packet headers be created, used, and modified? Examples include XCP [32], RCP [22, 49], PDQ [29], D^3 [53], and PERC [30].

Because datacenters are typically controlled by a single entity and involve workloads with measurable objectives, they are an ideal environment for experimentation. Nevertheless, most proposed resource management schemes have been evaluated only in simulation because no existing platform enables experimentation in router data planes with the desired level of programmability and usability. Unfortunately *simulations capture only part of the story*; they don't accurately model the nuances of real network stacks, NICs, and applications.

Previous work has explored two approaches to enabling data plane programmability. First, programmable hardware [7, 13, 34, 45, 47] enables users to program some data plane functions such as header manipulations. To date programmable switching chips have not provided sufficient flexibility to support many resource management schemes. This remains an active area of research [45, 46], but is not a viable option today. Furthermore, even if programmable hardware proves fruitful, users must replace their existing hardware to take advantage of it. Second, software routers [19, 21, 27, 33, 48] offer excellent flexibility, but provide insufficient router capacity for many applications, with the best results providing an aggregate throughput of only 30-40 Gbits/s.

In this paper we explore an alternative approach to programming resource management schemes in switches. We develop a system called **Flexplane**, which functions as an intermediate point between software routers and hardware. It remains as programmable as software routers, but provides substantially higher throughput. Flexplane does not match the performance of hardware, but is still well-suited for prototyping by researchers and students, evaluating new networking schemes with real-world operating systems and applications, and small-scale production.

The key idea in Flexplane is to move the software that implements the programmable data plane "off-path". We implement a user-specified scheme in a *centralized emulator*, which performs *whole-network emulation* in real time. The emulator maintains a model of the network topology in software, and users implement their schemes in the emulated routers. Before each packet is sent on the real network, a corresponding emulated packet traverses the emulated network and experiences behavior specified by the resource management scheme. Once this is complete, the real packet traverses the network without delay. Thus, applications observe a network that supports the

programmed scheme (Figure 1).

The rationale for this design is that resource management schemes are *data-independent*; they depend only on a small number of header fields, which typically comprise less than 1% of the total packet size.[1] Thus it suffices for resource management schemes to operate over short descriptions of packets, called *abstract packets*, rather than entire packets. This allows a software entity to support much higher aggregate throughput with the same network bandwidth.

We have implemented Flexplane, and have written seven different resource management schemes in it. We performed experiments to answer the question: *does off-path emulation over abstract packets provide a viable platform for experimentation with network resource management schemes?* Our results show that Flexplane provides accuracy, utility, and sufficient throughput for datacenter applications:

- **Accuracy:** Flexplane accurately reproduces the queue occupancies and flow completion times of schemes already supported in commodity hardware such as DropTail, RED, and DCTCP. For example, Flexplane matches the average normalized flow completion times for small flows in DCTCP to within 12% (§5.1).
- **Utility:** With Flexplane, users can implement a large number of schemes such as HULL, pFabric, etc. in a few dozen lines of code. They can use Flexplane to evaluate trade-offs between resource management schemes and to quickly tune protocol parameters for different link rates. Finally, users can experiment with real applications such as Spark and observe results that are not possible to observe in simulation, because they depend on the CPUs and network stacks of real endpoints (§5.2).
- **Throughput:** By limiting communication between cores, the Flexplane emulator scales nearly linearly to achieve 760 Gbits/s of throughput with 10 cores, for a topology of seven racks. This is about 20× as much as the RouteBricks software router while using one-third as many cores (§5.3).

2 Use Cases

In this section, we present three settings in which Flexplane can be useful and then describe the benefits of Flexplane over existing approaches.

Prototyping schemes. Network researchers frequently develop new resource management schemes. While prototyping, they need to test their schemes under realistic network conditions and to quickly iterate on their ideas.

[1] This contrasts with some router functions, such as encapsulation or encryption, which require the entire packet.

Evaluating schemes. Many people are eager to try out new schemes and to evaluate the performance impact of them on their applications. For example, network operators have a choice of what switch algorithms to run in their networks and may consider upgrading their hardware to support schemes that are newly available in ASICs (e.g., CONGA [9]). Before replacing their hardware, however, operators must test a new scheme with their specific applications to evaluate the potential benefits. As another example, students may want to implement, run, and measure resource management schemes to better understand them.

Programmable resource management in production. Although implementing resource management schemes in hardware provides the best performance, Flexplane can be used to achieve custom resource management in a small production network in the interim until a new scheme is supported in hardware. Fastpass [40] can be used for a similar purpose but today supports only one objective (max-min fairness). In contrast, Flexplane provides flexible APIs that enable users to express arbitrary new policies in only dozens of lines of code. For example, if one wishes to run applications on HULL in a production setting, Flexplane provides that ability over existing hardware.

2.1 Benefits over Existing Approaches

Simulation. Today, many resource management schemes (e.g., PDQ [29], pFabric [12], PERC [30], and LSTF [39]) are developed and tested exclusively using network simulators such as ns [6, 28] and OMNeT++ [51]. Flexplane provides flexibility similar to that of simluators, and also provides three additional benefits.

First, simulations are insufficient by themselves to demonstrate a scheme's effectiveness, because they do not capture the nuances of real network stacks and NICs. For many resource management schemes, implementation details matter, and endpoint optimizations can inform protocol design. For example, burstiness due to Interrupt Coalescing and other features led the authors of HULL [11] to add an endpoint pacer to their design. These NIC features are not easy to model in simulations, and without testing in a real network, it is hard to predict which optimizations matter. By using real machines as endpoints, Flexplane inherently captures these behaviors, allowing researchers to better understand how their protocols would work in practice.

Second, Flexplane enables users to evaluate resource management schemes using real applications. Realistic applications are not solely network bound; they involve a complex mixture of network I/O, disk I/O, and computation. These aspects of applications are hard to model in network simulations, but can be captured naturally by running Flexplane with unmodified applications. Further-

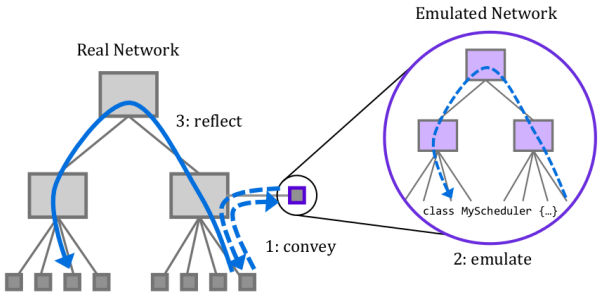


Figure 1: In Flexplane, endpoints *convey* their demands to the emulator in abstract packets, the emulator *emulates* packets traversing the network and experiencing the user-implemented scheme, and the results are *reflected* back onto the physical network. In the real network, dashed arrows represent an abstract packet, and the solid arrow represents the corresponding real packet.

more, with Flexplane users can evaluate real applications without porting them to simulation, which could be onerous for complex distributed systems (e.g., memcache or Spark). By enabling users to evaluate resource management schemes on real network hardware with real applications, Flexplane could mitigate risk for network operators and potentially enable greater adoption of new protocols.

Third, Flexplane experiments run in real time, completing much more quickly than simulations. For example, a 30ms ns-2 simulation with 100,000 flows took over 3 minutes to complete; this is 6000× slower than real time.

Software routers. Software routers are often used for research and experimentation (e.g., as in [54]), but Flexplane is a better approach for the use cases described above. Conducting large-scale experiments with software routers is infeasible because they provide insufficient throughput. Throughputs of 30-40 Gbits/s per router limit experiments to only a few servers per router with 10 Gbits/s links; Flexplane provides 20× as much throughput.

Programmable hardware. FPGAs and programmable switching chips do not provide the convenience of expressing schemes in C++, as in Flexplane, and do not enable experimentation in existing network infrastructure.

3 Design

Flexplane's goal is to enable a network to support a resource management scheme, even when the switch hardware does not provide it. More specifically, packets should arrive at their destinations with the timings and header modifications (e.g., ECN marks) that they would have in a hardware network that supported the desired scheme.

To achieve this goal, Flexplane implements the scheme in software in a single centralized multi-core server called the emulator, which endpoints consult before sending each packet.[2] The transmission of every packet in Flexplane involves three steps (Figure 1):

1. *Convey:* At the sending endpoint, Flexplane intercepts the packet before it is sent on the network. It constructs an *abstract packet* summarizing the key properties of the packet and sends it to the emulator (§3.1).
2. *Emulate:* The emulator models the entire network and emulates its behavior in real time. It delays and modifies abstract packets in the same way the corresponding real packets would be delayed and modified, if they traversed a hardware network implementing the same scheme (§3.2).
3. *Reflect:* As soon as an abstract packet exits the emulation, the emulator sends a response to the source endpoint. The endpoint immediately modifies the corresponding real packet (if necessary) and sends it, reflecting its fate onto the real network (§3.3).

These steps run in real time, with the result that packets experience the network-level behavior in the emulator rather than in the hardware network. While a real packet waits at an endpoint, its abstract packet traverses the emulated topology and encounters queueing and modifications there. When emulation is complete and the real packet traverses the real network, it will encounter almost no queueing and no modifications. Higher-layer protocols perceive that the packet is queued and modified by the resource management scheme implemented in the emulator. At the highest layer, datacenter applications experience a network that supports the emulated scheme.

3.1 Abstract Packets

An abstract packet concisely describes a chunk of data (e.g., a packet) to be sent onto the network. It includes the metadata required to route the packet through the emulated topology and the header fields that are accessed by routers running the chosen resource management scheme. All abstract packets include the source and destination addresses of the chunk of data, a unique identifier, and a flow identifier.

In addition, a scheme can include custom header fields in abstract packets, such as the type-of-service (DSCP) or whether the sender is ECN-capable. Flexplane provides a general framework that allows schemes to convey arbitrary packet information in abstract packets (§3.5). The ability to convey any information enables Flexplane to support existing resource management schemes as well as those that will be developed in the future. It also enables Flexplane to support schemes that require non-standard packet formats (e.g., pFabric [12]).

[2] Fastpass [40] previously introduced the idea of scheduling in a centralized entity at the granularity of individual packets. However, Fastpass schedules packets to achieve an explicit objective whereas Flexplane schedules packets to achieve the behavior of a distributed resource management scheme.

For simplicity of emulation, all abstract packets represent the same amount of data; this can consist of one large packet or multiple smaller packets in the same flow. In this paper, an abstract packet represents one maximum transmission unit (MTU) worth of packets (1500 bytes in our network).

Network bandwidth. Because communication between endpoints and the emulator occurs over the same network used to transmit real packets, it reduces the available network bandwidth (§6). However, abstract packets summarize only the essential properties of a chunk of data, and are quite small compared to the amount of data they represent. In a typical case, an abstract packet contains 2 bytes for each of its source, destination, flow, and unique identifier, and 4 bytes of custom data. These 12 bytes are less than 1% of the size of the corresponding MTU.

Abstract packets must be sent between endpoints and the emulator in real packets, adding additional overhead. However, multiple abstract packets can often be sent in the same real packet, e.g., when a single send() or sendto() call on a socket contains several packets worth of data. Abstract packets are also efficiently encoded into real packets; when an endpoint requests several packets in the same flow, the flow information is not included multiple times in the same packet to the emulator.

3.2 Emulation

The purpose of emulation is to delay and modify abstract packets just as they would be in a hardware network running the same resource management scheme. To achieve this, the emulator maintains an emulated network of routers and endpoints in software, configured in the same topology and with the same routing policies as the real network. When abstract packets arrive at the emulator, they are enqueued at their emulated sources. They then flow through the emulated topology, ending at their emulated destinations. As they flow through the emulated network, they may encounter queueing delays, acquire modifications from the routers they traverse (e.g., ECN marks), or be dropped, just as they would in a real network. The emulator is, effectively, a real-time simulator.

To simplify emulation, the emulator divides time into short timeslots. The duration of a timeslot is chosen to be the amount of time it takes a NIC in the hardware network to transmit the number of bytes represented by an abstract packet. Thus sending an abstract packet in the emulation takes exactly one timeslot. In each timeslot, the emulator allows each port on each emulated router and endpoint to send one abstract packet into the emulated network and to receive one abstract packet from the emulated network.[3] Timeslots simplify the task of ensuring that events in the emulation occur at the correct time. Instead of scheduling each individual event (e.g., packet transmission) at a specific time, the emulator only needs to ensure that each timeslot (which includes dozens of events) occurs at the right time. This contrasts with ns [28, 6] and other event-based simulators.

We assume that servers in the network support and run any part of a scheme that requires an endpoint implementation (e.g., the endpoint's DCTCP or XCP software). This frees the emulator from the burden of performing computation that can be performed in software at the endpoints instead. The emulator only emulates in-network functionality, from the endpoint NICs onwards, and the emulated endpoints act simply as sources and sinks to the rest of the emulation.

[3] We assume, for simplicity, that all links in the network run at the same rate. If this is not the case, multiple abstract packets could be sent and received per timeslot on the higher-bandwidth links.

3.3 Reflection

As soon as an abstract packet exits the emulation, the emulator sends a response to the source endpoint indicating how to handle the corresponding real packet on the real network. If the abstract packet was dropped during emulation, the emulator will instruct the real endpoint to drop the corresponding packet without sending it. Alternatively, if the abstract packet reached its emulated destination, the emulator will instruct the real endpoint to immediately modify the corresponding real packet (if necessary) and send it.

Modifying packets before sending them is required for correctness for schemes whose behavior depends upon received headers. For example, in ECN-based schemes [10, 11, 24, 50], routers set the ECN field in IP headers when congestion is experienced and the recipient must echo this congestion indication back to the sender. The receivers in explicit control schemes [22, 29, 30, 32, 53] similarly echo information about the network back to the senders.

Once a real packet has been modified, it will be sent onto the network. It will traverse the same path through the network that the corresponding abstract packet took through the emulation, because the emulator is configured to match the routing policies of the real network. The packet will not acquire more modifications in the real network, so it will arrive at its destination with the headers specified by the emulator.

3.4 Accuracy

To be a viable platform for experimentation, Flexplane must be accurate, meaning that its behavior predicts that of a hardware network. More precisely, we define *accuracy* as the similarity between results obtained by running an application over Flexplane with a given resource management scheme and results obtained by running the same application over a hardware network running the scheme.

Accuracy can be measured by comparing metrics at several different levels of the network stack. In section §5.1

Emulator pattern	int route(AbstractPkt *pkt) int classify(AbstractPkt *pkt, int port) enqueue(AbstractPkt *pkt, int port, int queue) AbstractPkt *schedule(int output_port)	return a port to enqueue this packet to return a queue at this port to enqueue this packet to enqueue this packet to the given port and queue return a packet to transmit from this port (or null)
Emulator generic	input(AbstractPkt **pkts, int n) output(AbstractPkt **pkts, int n)	receive a batch of n packets from the network output up to n packets into the network
Endpoints	prepare_request(sk_buff *skb, char *request_data) prepare_to_send(sk_buff *skb, char *allocation_data)	copy abstract packet data into request_data modify packet headers before sending

Table 1: Flexplane API specification. Flexplane exposes C++ APIs at the emulator for implementing schemes and C APIs at the physical endpoints, for reading custom information from packets and making custom modifications to packet headers. An sk_buff is a C struct used in the Linux kernel to hold a packet and its metadata.

we evaluate the extent to which Flexplane provides accuracy at the network and application levels, by analyzing metrics such as queue occupancies and flow completion time. Here we analyze Flexplane's accuracy at the granularity of individual packets, to better understand the ways in which Flexplane deviates from perfect accuracy.

To understand Flexplane's accuracy at the packet level, we compare the latency l' of a packet in a Flexplane network to the latency l of a packet in an identical hardware network that implements the same resource management scheme. For now, we assume that both networks consist of a single rack of servers. In the hardware network, the latency l experienced by a packet will be the sum of the unloaded delay u (the time to traverse the network when empty) and the queueing delay q: $l = u + q$. Note that the unloaded delay is the sum of the speed-of-light propagation delay (negligible in datacenter networks), processing delay at each switch, and the transmission delay (the ratio of the packet size to the link rate).

In the Flexplane network, the latency l' of a packet consists of the round-trip time r the abstract packet takes to the emulator and back, the emulated transmission delay t_e (the emulator does not model switching or propagation delays), the emulated queueing delay q_e, the time the real packet takes to traverse an unloaded network u, and any queueing delay it encounters in the real network, q'. For an emulator in the same rack, $r \leq 2u$ and the emulation ensures that $t_e < u$.

$$l' = r + t_e + q_e + u + q' < 4u + q' + q_e \qquad (1)$$

Flexplane does not guarantee zero queueing within the hardware network so q' may be nonzero, adding some jitter to l'. However, the emulation enforces bandwidth constraints, guaranteeing that there will be sufficient bandwidth to transmit all real packets admitted into the network to their destinations. This means that though there might be small amounts of transient queueing in the network, q' cannot grow indefinitely. In practice we find that q' is very small, approximately 8-12 µs on average for a fully utilized link (§5.1).

Emulation also contributes an additive delay to l'. Equation 1 estimates that for an unloaded network, the added latency in Flexplane is about three times the latency experienced in an equivalent hardware network, or less. However, our experiments indicate that in practice this additional latency is much smaller, about 1.3 times (§5.1). This difference is because a significant fraction of the unloaded delay u is spent in endpoint network stacks. The emulator uses kernel bypass, reducing r significantly below $2u$, and the emulation does not model endpoint network stacks, causing t_e to be significantly less than u. In addition, for loaded networks with congestion and queueing, these latency overheads contribute a smaller fraction of the overall latency, and Flexplane's latencies better match hardware.

Multiple racks. Flexplane supports multi-rack topologies. With multiple racks, the entire network is still scheduled using a single centralized emulator. As a result, round-trip times to the emulator and back (r) may vary slightly across endpoints. In addition, unloaded delays in the real network (u) will vary across packets that take different-length paths. To ensure that all packets experience the same delay overhead from Flexplane, endpoints delay packets so that the sum of r, u, and this additional delay equals the sum of the maximum values of r and u for any packet in the network. Thus with multiple racks, the delay imposed by Flexplane will increase from that described above, but will remain an additive factor that is constant across all packets. With larger networks q', the queueing delay in the real network, can also increase, adding more jitter to the end-to-end packet delay, l'.

In multi-rack topologies, real packets must follow the same paths as those taken by their emulated counterparts through the emulated network, in order to avoid causing congestion in the real network. When the implemented resource management scheme does not control packets' paths, this agreement can be achieved by configuring the routing policies in the emulated routers. For example, if ECMP is used in the real network with a known hash function, emulated routers can run ECMP with the same hash function. When the implemented resource management scheme does control packets' paths (as in CONGA [9] or DeTail [54]), packets can be made to follow the prescribed paths using tunneling or ECMP spoofing, as described in [40].

3.5 Flexplane APIs

Flexplane exposes simple APIs to users so that they can write their own resource management schemes. It decouples the implemented network schemes from the emulation framework, so that users only need to worry about the specific behavior of their scheme. The framework handles passing abstract packets between different emulated components and communicating with the physical endpoints.

Emulator APIs. To add a scheme to Flexplane, users can choose between two C++ APIs to implement at the emulator, as shown in Table 1. In the pattern API (the API we expect most users to follow), users implement four functions. These functions divide packet processing into routing (choosing an output port), classification (choosing amongst multiple queues for a given port), enqueuing (adding a packet to the chosen queue or dropping it), and scheduling (choosing a packet to dequeue from a port). The framework calls the `route`, `classify`, and `enqueue` functions, in order, for each packet arriving at a router, and calls the `schedule` function on each output port in a router once per timeslot. Each of these functions is implemented in a separate C++ class, enabling them to be composed arbitrarily. If the user wants more flexibility than the pattern API, we provide a more general API specified in the generic row of Table 1.

Endpoint APIs. Flexplane also provides a C API for users to read and modify packet headers at the physical endpoints. For schemes that require access to packet fields beyond the default abstract packet fields, the user specifies how to copy the desired custom information from the real packet into the abstract packet, by implementing the function `prepare_request`. Flexplane then conveys this data to the emulator along with the rest of the abstract packet so that it will be available to the emulated scheme.

Similarly, users may modify packet headers at the endpoints (e.g., to add ECN marks) by implementing the function `prepare_to_send`. This function is called for each packet immediately before it is sent on the real network and includes a pointer to the packet as well as an array of bytes from the abstract packet, populated by the resource management scheme in the emulator.

3.6 Scaling the Emulator with Multi-core

Supporting large networks with Flexplane is challenging because it requires high aggregate emulator throughput. The emulator uses multi-core to achieve this. It runs on a dedicated multi-core machine; as long as each additional core increases total throughput, the emulator can scale to large networks by using many cores.

Common approaches to multi-core packet-processing are ill-suited for Flexplane. For example, if Flexplane processed each abstract packet to completion on a single core (similar to RouteBricks [21]), all cores would contend for the shared state of emulated routers and endpoints, limiting scalability. Passing abstract packets down a linear pipeline (as in Fastpass [40]) does not work either, because different packets may access router state in different orders.

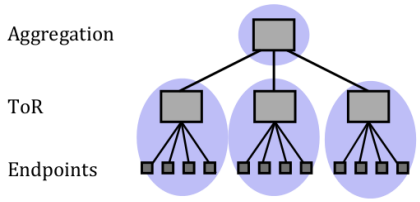


Figure 2: The emulator pins network components (grey) to CPU cores (purple) to avoid sharing state across cores.

Instead, Flexplane's multi-core architecture leverages the fact that networks of routers and endpoints are naturally distributed. Flexplane pins each component of the emulated network topology to a core and passes abstract packets from core to core as they traverse the topology. With this architecture, router state is never shared across cores and cores only communicate when their network components have packets to send to or receive from a different core. This limited communication between cores allows throughput to scale with the number of available cores (§5.3).

To assign emulated components to cores, the emulator distributes the routers amongst the available cores and assigns endpoints to the same core that handles their top-of-rack switch (Figure 2). The Flexplane framework handles inter-core communication with FIFO queues of abstract packets.

For large routers or computationally heavy schemes, one core may not provide sufficient throughput. Future work could explore splitting each router across multiple cores. Different output ports in a router typically have little or no shared state, so one option would be to divide a router up by its output ports and to process different ports on different cores.

Inter-core communication. Flexplane employs three strategies to reduce the overhead of inter-core communication. First, Flexplane maintains only *loose synchronization* between cores. Each core independently ensures that it begins each timeslot at the correct time using CPU cycle counters, but cores do not explicitly synchronize with each other. Tight synchronization, which we attempted with barriers, is far too inefficient to support high throughput. Second, Flexplane *batches* accesses to the FIFO queues so that all abstract packets to be sent on or received from a queue in a single timeslot are enqueued or dequeued together. This is important for reducing contention on shared queues. Third, Flexplane *prefetches* abstract packets the first time they are accessed on a core. This is possible because packets are processed in batches;

later packets can be prefetched while the core performs operations on earlier packets. This is similar to the group prefetching technique described in [31].

3.7 Fault Tolerance

Abstract packets. Flexplane provides reliable delivery of abstract packets both to and from the emulator. Because each abstract packet corresponds to a specific packet or group of packets (uniquely identified by their flow and sequential index within the flow), they are not interchangeable. For correct behavior, the emulator must receive an abstract packet for each real packet. If the physical network supports simple priority classes, then the traffic between endpoints and the emulator should run at the highest priority, making these packets unlikely to be dropped. Flexplane uses ACKs and timeouts to retransmit any abstract packets in the unlikely event that they are still dropped.

Emulator. We handle emulator fail-over in the same way as in Fastpass [40]. The emulator maintains only soft state so that a secondary emulator can easily take over on failure. The primary sends periodic watchdog packets to the secondary; when the secondary stops receiving them it takes over and begins emulation. Endpoints update the secondary with their pending abstract packets.

4 Implementation

We implemented Flexplane by extending Fastpass [40]. The Flexplane emulator uses the Intel Data Plane Development Kit (DPDK) [3] for low-latency access to NIC queues from userspace. Support for Flexplane at the endpoints is implemented in a Linux kernel module, which functions as a Linux `qdisc`, intercepting and queueing packets just before they are passed to the driver queue.

We have implemented and experimented with seven different schemes in Flexplane:

DropTail: DropTail queues with static per-queue buffers.

RED: Random Early Detection as described in [24], with and without ECN [23].

DCTCP: ECN marking at the switches, as in [10].

Priority queueing: strict priority queueing across at most 64 queues with classification based on the DSCP field.

Deficit round-robin: round-robin across at most 64 queues, as described in [44], with classification by flow.

HULL: phantom queues as described in [11]. We omit the hardware pacer at the endpoints.

pFabric: pFabric switches as described in [12]. We use standard TCP cubic at the endpoints, omitting the probe mode and other rate control optimizations. We use the remaining flow size as the priority for each packet.

Fastpass [40] relies on clock synchronization using the IEEE1588 Precision Time Protocol (PTP) to ensure that endpoints send packets at precisely the prescribed times. However, we found that it is sufficient to clock the transmissions of real packets using the arrivals of the corresponding abstract packets at the endpoint. This renders PTP unnecessary.

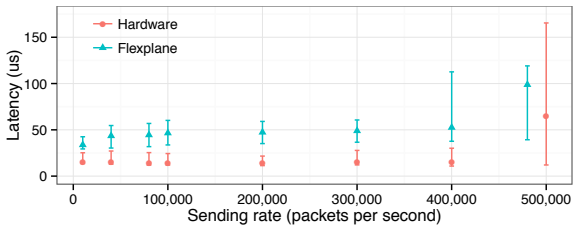


Figure 3: Packet latency under varying loads for one sender in Flexplane and the baseline hardware network. Points show medians; error bars show minimum and 99th percentile observed over 30 seconds.

5 Evaluation

We conduct experiments in a large production network on a single rack of 40 servers, each connected to a top-of-rack (ToR) switch. Each server has two CPUs, each with 20 logical cores, 32GB RAM, one 10 Gbits/s NIC, and runs the Linux 3.18 kernel. One server is set aside for running the emulator. The switch is a Broadcom Trident+ based device with 64x10 Gbits/s ports and 9 MBytes of shared buffer. The switch supports a few schemes such as WRED with ECN marking, which we disable in all experiments except in explicit comparisons for emulation accuracy. We use an MTU size of 1500 bytes, and we disable TCP segmentation offload (TSO) (§6.2).

Our experiments address the following questions:

1. **Accuracy**: How well does Flexplane predict the behavior of schemes already supported in hardware?
2. **Utility**: How easy is Flexplane to use and what forms of experimentation can it enable?
3. **Emulator throughput**: Does the Flexplane emulator provide sufficient throughput to support datacenter applications?

5.1 Accuracy

In this section, we evaluate how faithfully Flexplane predicts results obtained with hardware switches, in terms of latency for individual packets, throughput and queue occupancies in the network, and flow completion times observed by applications.

Latency. First we compare the latency of packets in Flexplane to that of packets running on bare hardware, in an uncongested network. For this experiment, one client sends MTU-sized UDP packets to a server in the same rack, using the `sockperf` utility [5]; the receiver sends back responses. We measure the RTT of the response packets at several different rates of packets sent per second, and estimate the one-way latency as the RTT divided

by two. We run DropTail both in Flexplane and on the hardware switch.

The results in Figure 3 demonstrate that the per-packet latency overhead of Flexplane is modest. Under the lightest offered load we measure (10,000 packets/s), the median latency in Flexplane is 33.8 µs, compared to 14.9 µs on hardware. As the load increases, the latency in Flexplane increases slightly due to the additional load on the kernel module in the sending endpoint. Flexplane is unable to meet the highest offered load (6 Gbits/s), because of the CPU overhead of the kernel module. Note that state-of-the-art software routers add latencies of the same order of magnitude for each hop, even without the added round-trip time to an off-path emulator: 47.6-66.4 µs [21] for a CPU-based software router; 30 µs [31] or 140-260 µs [27] for GPU-based software routers.

Throughput. Next we evaluate accuracy for bulk-transfer TCP, using network-level metrics: throughput and in-network queueing. In each experiment, five machines send TCP traffic at maximum throughput to one receiver.

We compare Flexplane to hardware for three schemes that our router supports: TCP-cubic/DropTail, TCP-cubic/RED, and DCTCP. We configure the hardware router and the emulator using the same parameters for each scheme. For DropTail we use a static per-port buffer size of 1024 MTUs. For RED, we use min_th=150, max_th=300, max_p=0.1, and weight=5. For DCTCP, we use an ECN-marking threshold of 65 MTUs, as recommended by its designers [10].

Flexplane achieves similar aggregate throughput as the hardware. All three schemes consistently saturate the bottleneck link, achieving an aggregate throughput of 9.4 Gbits/s in hardware, compared to 9.2-9.3 Gbits/s in Flexplane. This 1-2% difference in throughput is due to bandwidth allocated for abstract packets in Flexplane.

Queueing. During the experiment described above, we sample the total buffer occupancy in the hardware router every millisecond, and the emulator logs the occupancy of each emulated port at the same frequency.

Table 2 shows that Flexplane maintains similar queue occupancies as the hardware schemes. For DropTail it maintains high occupancies (close to the max of 1024) with large variations in occupancy, while for the other two schemes the occupancies are lower and more consistent. Flexplane does differ from hardware in that its occupancies tend to be slightly lower and to display more variation. We believe this is due to the effectively longer RTT in Flexplane. When the congestion window is reduced, the pause before sending again is longer in Flexplane, allowing the queues to drain more.

During the Flexplane experiments, the hardware queue sizes remain small: the mean is 7-10 MTUs and the 95th percentile is 14-22 MTUs. These numbers are small compared to the queue sizes in the emulator or in the hardware queues during the hardware experiments, and indicate that queueing in the hardware network does not significantly impact the accuracy of Flexplane (§3.4).

	Queue Occupancies (MTUs)			
	Hardware		Flexplane	
	median	σ	median	σ
DropTail	931	73.7	837	98.6
RED	138	12.9	104	32.5
DCTCP	61	4.9	51	13.0

Table 2: Flexplane achieves similar queue occupancies and standard deviations in occupancies (σ) as hardware.

Flow completion time. Next we evaluate Flexplane's accuracy at the application level in terms of flow completion time (FCT). We run an RPC-based application in which four clients repeatedly request data from 32 servers. The size of the requested data is determined by an empirical workload derived from live traffic in a production datacenter that supports web search (first presented in [10]). It includes a mixture of flows of different sizes. 53% of the flows are small flows of less than 100KB, but 37% of the bytes come from large flows of 10MB or larger. Request times are chosen by a Poisson process such that the clients receive a specified load between 10% and 80%. We normalize the FCT for each flow to the average FCT achieved by a flow of the same size, in an unloaded network, when flows are requested continuously.

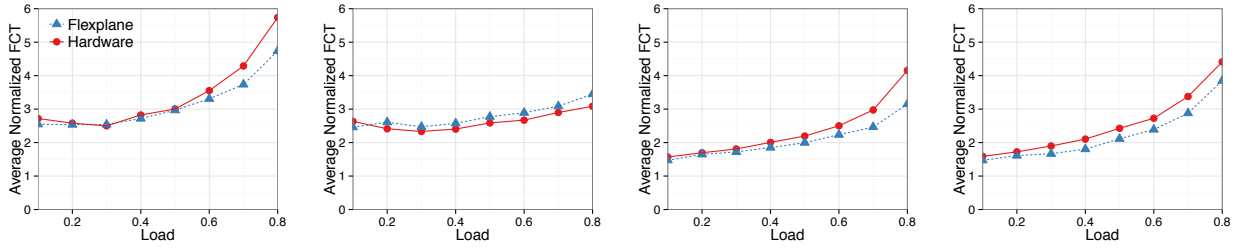
We run this application for DropTail and DCTCP, in Flexplane and in the hardware network. Figure 4 shows the average normalized FCTs. For both small flows and large flows, results with Flexplane closely match results obtained with a hardware network. For loads up to 60% with both schemes, Flexplane estimates average normalized FCTs of hardware to within 2-8% for small flows and 3-14% for large flows. Accuracy decreases slightly for higher loads of 70% and 80%, but remains within 18% for small flows and 24% for large flows.

5.2 Flexplane Utility

In this section, we evaluate the utility of Flexplane. We study how easy it is to write new schemes in Flexplane and provide four examples of how Flexplane can be useful.

Ease of use. To demonstrate the simplicity of implementation, we show the key portions of the source code for priority queueing scheduling in Figure 5. Most schemes require only a few dozen lines of code to implement, as shown in Table 3. pFabric requires significantly more code than other schemes because it does not maintain packets in FIFO order between the enqueue and dequeue stages; 170 of the 251 lines of code are used to implement custom queueing.

Parameter tuning. Flexplane enables users to quickly tune protocol parameters to accommodate different net-



(a) DropTail, (0, 100KB] **(b)** DCTCP, (0, 100KB] **(c)** DropTail, (10MB, ∞) **(d)** DCTCP, (10MB, ∞)

Figure 4: Flexplane closely matches the average normalized flow completion times of hardware for DropTail and DCTCP. The left two graphs show results for small flow sizes; the right two graphs show results for large flow sizes.

```
class PriorityScheduler : public Scheduler {
public:
  AbstractPkt* PriorityScheduler::schedule(uint32_t
       port) {
    /* get the mask of non-empty queues */
    uint64_t mask = m_bank->non_empty_qmask(port);

    uint64_t q_index;
    /* bsfq: find the first set bit in mask */
    asm("bsfq %1,%0":"=r"(q_index):"r"(mask));

    return m_bank->dequeue(port, q_index);
   }
private:
  PacketQueueBank *m_bank;
}
```

Figure 5: Source code for a priority scheduler in Flexplane over ≤ 64 queues. A PacketQueueBank stores packets between the calls to `enqueue` and `schedule`.

scheme	LOC
drop tail queue manager	39
RED queue manager	125
DCTCP queue manager	43
priority queueing scheduler	29
round robin scheduler	40
HULL scheduler	60
pFabric QM, queues, scheduler	251

Table 3: Lines of code (LOC) in the emulator for each resource management scheme.

works. For example, the authors of HULL [11] conducted evaluations using a testbed with 1 Gbits/s links; we use Flexplane to tune HULL's parameters to fit our 10 Gbits/s network. We use the recommended phantom queue drain rate of 95% of the link speed (9.5 Gbits/s). The HULL authors use a 1 KB marking threshold in a 1 Gbits/s network, and suggest a marking threshold of 3-5 KB for 10 Gbits/s links. We found, however, that throughput degraded significantly with a 3 KB marking threshold, achieving only 5 Gbits/s total with four concurrent flows. We therefore increased the marking threshold until our achieved throughput was 92% of the drain rate (this is what [11] achieves with their parameters); the resulting threshold is 15 KB. Flexplane helped us conduct this parameter search quickly and effectively.

Evaluating trade-offs. In this example, we demonstrate how one might use Flexplane to evaluate the performance

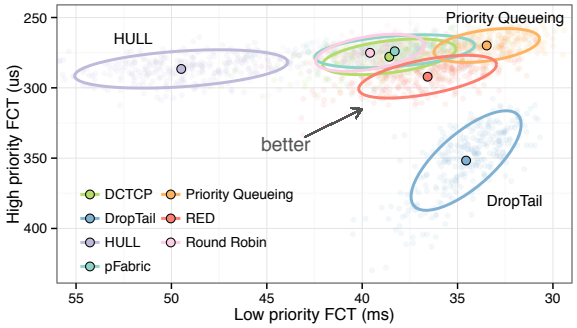
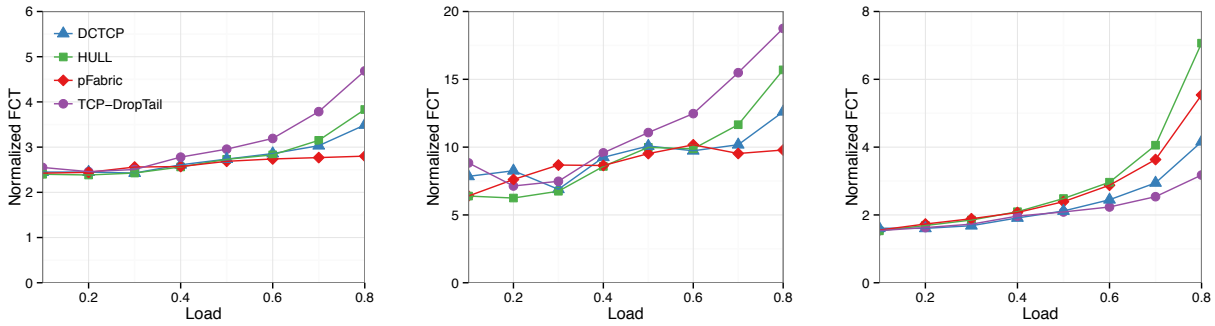


Figure 6: Flexplane enables users to explore trade-offs between different schemes. Large points show averages over the entire experiment, faded points show averages over 1s, and ellipses show one standard deviation. Note the flipped axes.

of a specific application with different resource management schemes. We do not argue that any scheme is better than any other, but instead demonstrate that there are trade-offs between different schemes (as described in [47]), and that Flexplane can help users explore these trade-offs.

We use an RPC-based workload and consider the trade-off that schemes make between performance for short flows and performance for long flows. In the experiment, four clients repeatedly request data from 32 servers. 80% of the requests are short 1.5 KB "high priority" requests, while the remaining 20% are 10 Mbyte "low priority" requests. Request times are chosen by a Poisson process such that the client NICs are receiving at about 60% of their maximum throughput. We evaluate the schemes discussed in §5.1, as well as TCP-cubic/per-flow-DRR, TCP-cubic/priority-queueing, HULL, and pFabric.

Figure 6 shows the trade-off each scheme makes on this workload. With DropTail, large queues build up in the network, leading to high flow completion times for the high-priority requests. However, DropTail senders rarely cut back their sending rates and therefore achieve good FCTs for the long requests. At the other end of the spectrum, HULL's phantom queues cause senders to decrease their sending rates early, leading to unutilized bandwidth and worse performance for the low priority flows; the high priority flows achieve relatively good performance because they encounter little queueing in the network.



(a) (0, 100KB]: Average **(b)** (0, 100KB]: 99th percentile **(c)** (10MB, ∞): Average

Figure 7: Normalized flow completion times for the web search workload, for four different schemes run in Flexplane. Note the different y axes.

Priority queueing performs well on this simple workload, achieving good performance for both flow types. A network operator could use these results to determine what scheme to run in their network, depending on how they value performance of high priority flows relative to low priority flows.

Real applications. In addition to enabling experimentation with network-bound workloads like the one above, Flexplane enables users to evaluate the performance impact of different resource management schemes on real applications whose performance depends on both network and computational resources. We consider two applications that perform distributed computations using Spark [1]. The first uses block coordinate descent [2] to compute the optimal solution to a least squares problem; this is a staple of many machine learning tasks. The second performs an in-memory sort [4]. For this experiment, we use a small cluster of 9 machines (1 master and 8 workers), each with 8 cores, connected via a single switch with 1 Gbit/s links. We use Flexplane to run each application with DropTail, DCTCP, and HULL.

Table 4 shows that different Spark applications are affected in different ways by a change in resource management scheme. The sort application, which includes multiple waves of small tasks and small data transfers, shows small improvements in completion time, relative to DropTail, when run with DCTCP or HULL. In contrast, coordinate descent takes 4.4% longer to complete when run with DCTCP, and 29.4% longer when run with HULL. This is because this application sends data in a small number of bulk transfers whose throughput is degraded by HULL's, and to a lesser extent DCTCP's, more aggressive responses to congestion. Flexplane enabled us to quickly evaluate the impact of a change in resource management scheme on these real-world applications. Because these applications spend much of their time performing computation (>75%), it is not possible to accurately conduct this experiment in a network simulator today.

Reproducible research. Here we demonstrate how experiments that researchers conducted in simulation in the past can be conducted on a real network with Flexplane, and how results in a real network might differ from those in simulation. To do so, we recreate an experiment that has been conducted in several other research papers [12, 14, 26]. We use the same network configuration and workload as in the flow completion time experiment in §5.1; this is the same workload used in prior work.

	% Change in Completion Time	
	Coordinate descent	Sort
DCTCP	+4.4%	-4.8%
HULL	+29.4%	-2.6%

Table 4: Percent change in completion time of two Spark applications when run with DCTCP or HULL, relative to when run with DropTail.

Figure 7 shows the results of running this workload for DropTail, DCTCP, HULL, and pFabric, in Flexplane, at loads ranging from 10% to 80%. We present the average and 99th percentile normalized flow completion time for small flows, and the average normalized flow completion time for large flows, as in prior work.

We observe the same general trends as in prior work. For the small flows, DropTail performs the worst, with performance degrading significantly at the highest loads and at the 99th percentile. In contrast, pFabric maintains good performance for small flows, even at high load and at the tail. For large flows, DCTCP and DropTail maintain the best performance, while HULL and pFabric degrade significantly at loads of 70%-80%. For HULL, this is because the required bandwidth headroom begins to significantly limit large flows. For pFabric, performance degrades at high load because short queues cause many packets to be dropped. This may be exacerbated by the fact that we do not use all TCP modifications at the endpoints, including the probe mode (which is particularly important at high load).

Our results demonstrate an unexpected phenomenon. One would expect that under low load (e.g., 10%), small flows would achieve a normalized FCT close to

1; previous simulation results have corroborated this intuition [12, 26]. In contrast, our results show that the average normalized FCTs across all schemes begin at around 2.5, even under the lightest load. These results obtained in Flexplane agree with those obtained on the hardware network, for DropTail and DCTCP (Figure 4).

This unexpected behavior is due to the properties of real endpoint network stacks. In real endpoints, application-layer latency depends on the rate at which packets are sent; when packets are sent at a high enough rate, the latency decreases significantly. For example, in our network, the `ping` utility reports average ping latencies of 77 µs when pings are sent every 2 ms; this decreases to 14 µs when pings are sent every 50 µs. Because many of the bytes in this workload belong to large flows, the number of queries per second is relatively small (513 per second to saturate a 10 Gbits/s NIC). The result is that, under most loads, packets are not sent at a high enough rate for small flows to achieve the ultra-low latency achieved when flows are requested continuously; their normalized FCTs are thus much higher than 1. Large flows still approach normalized FCTs of 1 because the FCT is dominated by the transmission time. This behavior would be hard to capture accurately in simulation, but is automatically captured with Flexplane.

5.3 Emulator Throughput

The aggregate throughput of the Flexplane emulator determines the size of network and the types of applications that Flexplane can support. In this section we evaluate how emulator throughput scales with the number of cores and how it varies across resource management schemes.

Workload. For all throughput experiments, we generate a synthetic network load using additional cores on the emulator machine. Sources and destinations are chosen uniformly at random. Timings obey Poisson arrivals and we vary the mean inter-arrival time to produce different network loads. We run an automated stress test to determine the maximum sustainable throughput for a given configuration. It begins with low load and periodically adjusts the load, increasing it as long as all cores are able to sustain it, and decreasing it when a core falls behind. We report the total throughput observed over the last 30 seconds, aggregated over all routers in the topology. We conduct experiments on a 2.4 GHz Intel Xeon CPU E7-8870 with 10 cores and 32GB of RAM.

Scaling with number of cores. We consider several variants on a simple datacenter network topology. Each topology includes racks of 32 endpoints connected via a ToR switch; ToRs are connected by a single aggregation (Agg) switch. We assign the Agg switch to its own CPU core, and assign each ToR and its adjacent endpoints to another core, as shown in Figure 2. As we vary the number of racks in the topology, we also vary the fraction of traffic whose source and destination are in different racks so that in any given configuration, all routers process the same amount of network throughput. All routers run DropTail.

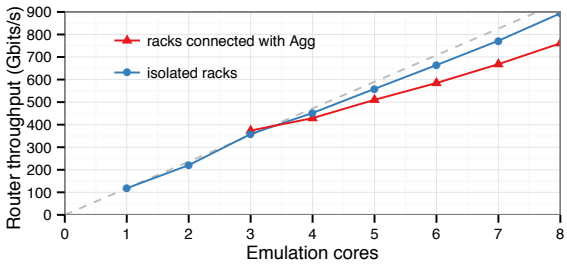


Figure 8: Maximum throughput achieved by the Flexplane emulator for different numbers of emulation cores. The grey dashed line indicates linear scaling based on the throughput achieved with a single emulation core.

The largest topology we measure has seven racks and achieves a total throughput of 761 Gbits/s. Thus, for example, the emulator could support a network of 224 servers sending at an average rate of 2.5 Gbits/s or a network of 56 servers sending continuously at 10 Gbits/s. This is sufficient throughput for small scale datacenter applications with high network utilization, or medium scale applications with lower network utilization.

Figure 8 shows how throughput scales with different numbers of racks in the topology and correspondingly with different numbers of emulation cores. When racks are connected with an Agg switch (red line), throughput falls short of linear scaling (grey line), but each additional core still provides about 77 Gbits/s of additional throughput. To understand the shortfall from linear scaling, we also show the throughput achieved by a simplified topology of isolated racks in which there is no Agg switch and all traffic is intra-rack (blue line). With this topology, throughput scales almost linearly, achieving 112 Gbits/s of added throughput per core on average, 95% of the 118 Gbits/s achieved by a single core. Thus only a small portion (at most 15%) of the throughput shortfall with connected racks is due to unavoidable contention between cores for shared resources such as the L3 cache. The majority of the shortfall is due to communication with the extra Agg core; a more sophisticated mechanism for inter-core communication might help reduce this shortfall.

Throughput across schemes. Figure 9 shows the maximum throughput achieved by each of the schemes we implemented in Flexplane, for a simple topology with one rack of 32 endpoints. DCTCP and DropTail achieve the highest throughput, about 120 Gbits/s. Most other schemes are only slightly more complex and achieve similar throughputs. The most complex scheme, pFabric, achieves 85 Gbits/s, demonstrating that even complex schemes can achieve sufficient throughput in Flexplane.

Comparison with software routers. Flexplane outperforms existing software routers in terms of individual

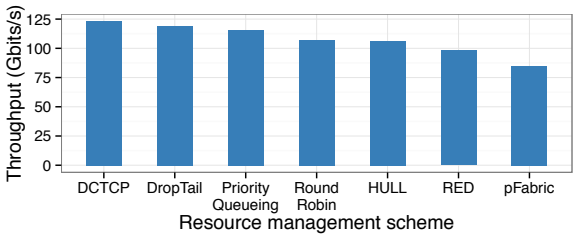


Figure 9: Maximum throughput achieved by the emulator for different resource management schemes with one rack of 32 machines.

router capacity. When emulating a single router with 32 endpoints, Flexplane achieves a total throughput of 118 Gbits/s, compared to a maximum of 35 Gbits/s in RouteBricks [21] and 40 Gbits/s in PacketShader [27], with minimal forwarding.

The difference between Flexplane and previous approaches is even more pronounced when we compare throughput per clock cycle, to normalize for different numbers of cores used and different clock frequencies. In its largest configuration, RouteBricks achieved 35 Gbits/s of throughput with 32 2.8 GHz cores. In comparison, our 7 rack configuration achieves a total router throughput of 761 Gbits/s with 10 2.4 GHz cores (8 emulation cores plus 2 stress test cores). This amounts to 81 times as much throughput per clock cycle in Flexplane.

The difference arises for two reasons. Because Flexplane only processes abstract packets, its memory footprint is relatively small; all of the memory used in the stress test fits in the 30 MB L3 cache shared by the 10 cores. In contrast, RouteBricks applications accessed 400-1000 bytes from memory for each 64-byte packet, likely degrading throughput. In addition, Flexplane performs no processing on its forwarding path involving data or header manipulation, leaving the hardware routers to handle that and focusing only on resource management functions.

6 Discussion

6.1 Overheads

Flexplane consumes some network resources; in this section, we quantify these overheads.

Network bandwidth. Communication between endpoints and the emulator consumes some of the network bandwidth. We measured the total throughput loss due to this communication (including abstract packets, acknowledgements, and retransmissions) to be only 1-2% of the total achievable throughput on a 10 Gbits/s link (§5.1).

Emulator NICs. To support larger networks, the emulator may need multiple NICs. Assuming 10 Gbits/s NICs and that traffic to the emulator is about 1-2% of all network traffic, the emulator needs one NIC for every 500-1000 Gbits/s of network throughput provided.

Emulator CPUs. The emulator requires about one communication core per 550 Gbits/s of network throughput and one emulation core per 77 Gbits/s of throughput. This means that every 550 Gbits/s of network throughput requires about 8 cores in the emulator.

6.2 Limitations

Though Flexplane has many uses (§5.2), it has a few limitations. First, Flexplane cannot scale to support arbitrarily large networks, because of throughput limitations at the emulator. Our 10-core emulator supports up to 760 Gbits/s of throughput (§5.3), sufficient for experiments spanning a few racks with 10 Gbits/s links, but insufficient for large-scale experiments involving dozens of racks. Second, in order to provide high performance, Flexplane maintains a fixed abstract packet size (§3.1); this may degrade accuracy with schemes whose behavior depends on packet size (e.g., fair queueing [20]) under workloads with diverse packet sizes. Third, because Flexplane adds some latency overhead (§3.4), it is not suitable for experimentation with schemes that display drastically different behavior with small changes in network latency. Finally, in order to faithfully emulate in-network behavior, Flexplane requires the ability to control the transmission time of each individual packet. This means that TCP segmentation offload (TSO) must be disabled in order to use Flexplane. Without TSO, many network stacks are unable to saturate high speed links (e.g., 10 Gbits/s and faster) with a single TCP connection; a multi-core stack may overcome this limitation.

7 Related Work

Existing approaches to programming and experimenting with resource management schemes fall into three broad categories: simulation (as discussed in §2.1), programmability in software, and programmable hardware.

7.1 Programmability in Software

Software routers. Software routers such as Click [18, 33], Routebricks [21], PacketShader [27] and GSwitch [52] process packets using general-purpose processors or GPUs, providing similar flexibility to Flexplane. However, Flexplane requires much less CPU and network bandwidth – a fiftieth to a hundredth of each – to achieve the same router throughput. Other software approaches [8, 35, 43] supplement hardware with programmable elements, but face the same throughput limitations because they must process packets "on-path" with CPUs for full flexibility.

End host approaches. Eden provides a programmable data plane for functions that can be implemented purely at the end hosts [15]. Unlike Flexplane, it cannot support schemes that require in-network functions beyond priority queueing, such as D^3 [53], PDQ [29], or pFabric [12].

7.2 Programmable Hardware

Several approaches such as NetFPGA [34], RiceNIC [42], CAFE [36], Chimpp [41], Switchblade [13], and [47] use FPGAs to enable programmable header manipulations or resource management. Though these approaches provide higher performance than software approaches, programming an FPGA is typically much harder than programming Flexplane in C++. In addition, a network must be equipped with FPGAs to benefit from such approaches.

Other work targets programmable switching chips. The P4 language [17] aims to provide a standard header manipulation language to be used by SDN control protocols like OpenFlow [38]. However, P4 does not yet provide the primitives necessary to support many resource management schemes, and, as a domain-specific language, P4 falls short of the usability of C++. The Domino language [45] allows users to express resource management schemes in a C-like language, which the Domino compiler then compiles to programmable switches. Domino approaches the flexibility of Flexplane, but users must upgrade to new programmable switches in order to reap the benefits. In contrast, Flexplane provides a way to experiment in existing network infrastructure.

8 Conclusion

In this paper we presented Flexplane, an experimentation platform for users to program resource management algorithms in datacenter networks. We demonstrated that Flexplane accurately reproduces the behavior of schemes already supported in hardware, sustains aggregate throughput of up to 760 Gbits/s with a 10-core implementation, and enables experimentation with schemes not yet supported in commodity routers. Flexplane offers a practical alternative to simulation for researchers, a way of evaluating new protocols for network operators, and a platform for experimenting in real networks for students.

Acknowledgements

We thank Omar Baldonado and Sanjeev Kumar of Facebook for their enthusiastic support of this collaboration, Vadim Balakhovski of Mellanox for his generous assistance with Mellanox drivers and equipment, Devavrat Shah for useful discussions, Kay Ousterhout and Shivaram Venkataraman for their assistance in running Spark applications, and our shepherd Anja Feldmann and the NSDI reviewers for their useful feedback. Ousterhout was supported by an NSF Fellowship and a Hertz Foundation Fellowship. Perry was supported by a Facebook Fellowship. This work was funded in part by NSF Grant CNS-1526791. We thank the industrial members of the MIT Center for Wireless Networks and Mobile Computing for their support and encouragement.

References

[1] Apache Spark. http://spark.apache.org/.

[2] Distributed Matrix Library. https://github.com/amplab/ml-matrix.

[3] DPDK Boosts Packet Processing, Performance, and Throughput. http://www.intel.com/go/dpdk.

[4] MemorySortJob. https://github.com/kayousterhout/spark-1/blob/monoDeadline/examples/src/main/scala/org/apache/spark/examples/monotasks/MemorySortJob.scala.

[5] Network Benchmarking Utility. https://github.com/mellanox/sockperf.

[6] The Network Simulator - ns-2. http://www.isi.edu/nsnam/ns/index.html.

[7] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson. The Next Generation of Intel IXP Network Processors. *Intel Technology Journal*, 6(3):6–18, 2002.

[8] M. Al-Fares, R. Kapoor, G. Porter, S. Das, H. Weatherspoon, B. Prabhakar, and A. Vahdat. NetBump: User-extensible Active Queue Management with Bumps on the Wire. In *ANCS*, 2012.

[9] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, G. Varghese, et al. CONGA: Distributed Congestion-Aware Load Balancing for Datacenters. In *SIGCOMM*, 2014.

[10] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[11] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is More: Trading a little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI*, 2012.

[12] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-Optimal Datacenter Transport. In *SIGCOMM*, 2013.

[13] M. B. Anwer, M. Motiwala, M. b. Tariq, and N. Feamster. SwitchBlade: A Platform for Rapid Deployment of Network Protocols on Programmable Hardware. In *SIGCOMM*, 2010.

[14] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *NSDI*, 2015.

[15] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea. Enabling End-host Network Functions. In *SIGCOMM*, 2015.

[16] J. C. R. Bennett and H. Zhang. Hierarchical Packet Fair Queueing Algorithms. In *SIGCOMM*, 1996.

[17] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM CCR*, July 2014.

[18] B. Chen and R. Morris. Flexible Control of Parallelism in a Multiprocessor PC Router. In *ATC*, 2001.

[19] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next Generation Routers. In *SIGCOMM*, 1998.

[20] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *SIGCOMM CCR*, Sep. 1989.

[21] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *SOSP*, 2009.

[22] N. Dukkipati and N. McKeown. Why Flow-Completion Time is the Right Metric for Congestion Control. *SIGCOMM CCR*, Jan. 2006.

[23] S. Floyd. TCP and Explicit Congestion Notification. *SIGCOMM CCR*, Oct. 1994.

[24] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4), Aug. 1993.

[25] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ICFP*, 2011.

[26] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues Don't Matter When You Can JUMP Them! In *NSDI*, 2015.

[27] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-Accelerated Software Router. In *SIGCOMM*, 2010.

[28] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena. Network Simulations with the ns-3 Simulator. *SIGCOMM demonstration*, 2008.

[29] C. Y. Hong, M. Caesar, and P. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *SIGCOMM*, 2012.

[30] L. Jose, L. Yan, M. Alizadeh, G. Varghese, N. McKeown, and S. Katti. High Speed Networks Need Proactive Congestion Control. In *HotNets*, 2015.

[31] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen. Raising the Bar for Using GPUs in Software Packet Processing. In *NSDI*, 2015.

[32] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *SIGCOMM*, 2002.

[33] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *TOCS*, 18(3):263–297, 2000.

[34] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. NetFPGA–An Open Platform for Gigabit-rate Network Switching and Routing. In *IEEE International Conference on Microelectronic Systems Education*, 2007.

[35] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *NSDI*, 2011.

[36] G. Lu, Y. Shi, C. Guo, and Y. Zhang. CAFE: A Configurable Packet Forwarding Engine for Data Center Networks. In *PRESTO*, 2009.

[37] P. E. McKenney. Stochastic Fairness Queuing. In *INFOCOM*, 1990.

[38] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR*, Apr. 2008.

[39] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker. Universal Packet Scheduling. In *NSDI*, 2016.

[40] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *SIGCOMM*, 2014.

[41] E. Rubow, R. McGeer, J. Mogul, and A. Vahdat. Chimpp: A Click-based Programming and Simulation Environment for Reconfigurable Networking Hardware. In *ANCS*, 2010.

[42] J. Shafer and S. Rixner. RiceNIC: A Reconfigurable Network Interface for Experimental Research and Education. In *ExpCS*, 2007.

[43] A. Shieh, S. Kandula, and E. G. Sirer. SideCar: Building Programmable Datacenter Networks without Programmable Switches. In *HotNets*, 2010.

[44] M. Shreedhar and G. Varghese. Efficient Fair Queuing Using Deficit Round-Robin. *IEEE/ACM Transactions on Networking*, 4(3), June 1996.

[45] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *SIGCOMM*, 2016.

[46] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable Packet Scheduling at Line Rate. In *SIGCOMM*, 2016.

[47] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan. No Silver Bullet: Extending SDN to the Data Plane. In *HotNets*, 2013.

[48] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a Robust Software-Based Router Using Network Processors. In *SOSP*, 2001.

[49] C. Tai, J. Zhu, and N. Dukkipati. Making Large Scale Deployment of RCP Practical for Real Networks. In *INFOCOM*, 2008.

[50] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-Aware Datacenter TCP (D^2TCP). *SIGCOMM*, 2012.

[51] A. Varga et al. The OMNeT++ Discrete Event Simulation System. In *ESM*, 2001.

[52] M. Varvello, R. Laufer, F. Zhang, and T. Lakshman. Multi-Layer Packet Classification with Graphics Processing Units. In *CoNEXT*, 2014.

[53] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM*, 2011.

[54] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. *SIGCOMM*, 2012.

I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades

Syed Akbar Mehdi[1], Cody Littley[1], Natacha Crooks[1], Lorenzo Alvisi[1,4], Nathan Bronson[2], and Wyatt Lloyd[3]

[1]UT Austin, [2]Facebook, [3]USC, [4]Cornell University

Abstract

We describe the design, implementation, and evaluation of Occult (**O**bservable **C**ausal **C**onsistency **U**sing **L**ossy **T**imestamps), the first scalable, geo-replicated data store that provides causal consistency to its clients without exposing the system to the possibility of *slowdown cascades*, a key obstacle to the deployment of causal consistency at scale. Occult supports read/write transactions under PC-PSI, a variant of Parallel Snapshot Isolation that contributes to Occult's immunity to slowdown cascades by weakening how PSI replicates transactions committed at the same replica. While PSI insists that they all be totally ordered, PC-PSI simply requires total order Per Client session. Nonetheless, Occult guarantees that all transactions read from a causally consistent snapshot of the datastore without requiring any coordination in how transactions are asynchronously replicated.

1 Introduction

Causal consistency [7] appears to be ideally positioned to respond to the needs of the sharded and geographically replicated data stores that support today's large-scale web applications. Without imposing the high latency of stronger consistency guarantees [30, 38], it can address many issues that eventual consistency leaves unresolved. This brings clear benefits to users and developers: causal consistency is all that is needed to preserve operation ordering and give Alice assurance that Bob, whom she had defriended before posting her Spring-break photos, will not be able to access her pictures, even though Alice and Bob access the photo-sharing application using different replicas [13, 20, 39]. Yet, causal consistency has not seen widespread industry adoption.

This is not for lack of interest from the research community. In the last few years, we have learned that no guarantee stronger than real-time causal consistency can be provided in a replicated data store that combines high availability with convergence [43], and that, conversely, it is possible to build convergent causally-consistent data stores that can efficiently handle a large number of shards [10, 14, 27, 28, 39, 40].

We submit that industry's reluctance to deploy causal consistency is in part explained by the inability of its current implementations to comply with a basic commandment for scalability: do not let your performance be determined by your slowest component. In particular, current causal systems often prevent a shard in replica R from applying a write w until all shards in R have applied all the writes that causally precede w. Hence, a slow or failed shard (a common occurrence in any large-enough deployment) can negatively impact the entire system, delaying the visibility of updates across many shards and leading to growing queues of delayed updates. As we show in Section 2, these effects can easily snowball to produce the "slowdown cascades" that Facebook engineers recently indicated [8] as one of the key challenges in moving beyond eventual consistency.

This paper presents Occult (**O**bservable **C**ausal **C**onsistency **U**sing **L**ossy **T**imestamps), the first geo-replicated and sharded data store that provides causal consistency to its clients without exposing the system to slowdown cascades. To make this possible, Occult shifts the responsibility for the enforcement of causal consistency from the data store to its clients. The data store makes its updates available as soon as it receives them, and causal consistency is enforced on reads only for those updates that clients are actually interested in observing. In essence, Occult decouples the rate at which updates are applied from the performance of slow shards by optimistically *rethinking the sync* [48]: instead of enforcing causal consistency as an invariant of the data store, through its read-centric approach Occult appears to applications as *indistinguishable* from a system that does.

Because it never delays writes to enforce consistency, Occult is immune from the dangers of slowdown cascades. It may, however, delay read operations from shards that are lagging behind to ensure they appear consistent with what a user has already seen. We expect such delays to be rare in practice because a recent study of Facebook's eventually-consistent production system found that fewer than six out of every million reads were not causally consistent [42]. Our evaluation confirms this. We find that our prototype of Occult, when compared with the eventually-consistent system (Redis Cluster) it is derived from, increases the median latency by only 50μs, the 99th percentile latency by only 400μs for a read-heavy workload (4ms for a write-heavy workload), and reduces throughput by only 8.7% for a read-heavy workload (6.9% for a write-heavy workload).

Occult's read-centric approach, however, raises a thorny technical issue. Occult requires clients to determine how their

local state depends on the state of the *entire* data store; such global awareness is unnecessary in systems that implement causal consistency within the data store, where simply tracking the immediate predecessors of a write is enough to determine when the write should be applied [39]. In principle, it is easy to use vector clocks [29, 44] to track causal dependencies at the granularity of objects or shards. However, their overhead at the scale that Occult targets is prohibitive. Occult instead uses *causal timestamps* that, by synthesizing a variety of techniques for compressing dependency information, can achieve high accuracy (reads do not stall waiting for updates that they do not actually depend on) at low cost. We find that 24-byte timestamps suffice to achieve an accuracy of 99.6%; 8 more bytes give an accuracy of 99.96%.

Causal timestamps also play a central role in Occult's support for scalable read-write transactions. Transactions in Occult operate under a variant of Parallel Snapshot Isolation [55]. Occult ensures that all transactions always *observe* a consistent snapshot of the system, even though the datastore no longer evolves through a sequence of monotonically increasing consistent snapshots. It uses causal timestamps to not only track transaction ordering but also atomicity (by making writes of a transaction *causally dependent on each other*). This novel approach is key to the scalability of Occult's transactions and their immunity to slowdown cascades. The responsibility for commit is again shifted to the client, which uses causal timestamps to detect if a transaction has observed an inconsistent state due to an ordering or atomicity violation and, if so, aborts it. Committed writes instead propagate asynchronously to slaves, allowing the commit logic to scale independently of the number of slave replicas.

In summary, the contributions of this paper include:

- A novel and light-weight read-centric implementation of causal consistency. By shifting enforcement to the clients, it ensures that they never observe non-causal states, while placing no restrictions on the data store.
- A new transactional isolation level called Per-Client Parallel Snapshot Isolation (PC-PSI), a variant of PSI, that contributes to Occult's immunity to slowdown cascades by weakening how PSI replicates transactions committed at the same replica.
- A novel scalable protocol for providing PC-PSI that uses causal timestamps to enforce both atomicity and transaction ordering and whose commit latency is independent of the number of replicas in the system.
- An implementation and evaluation of Occult, the first causally consistent store that implements these ideas and is immune to slowdown cascades.

2 Motivation

Causal consistency ensures that clients observe their own updates and read from a state that includes all operations that they have previously observed. This promise aims for a sweet spot in the debate on the guarantees that a sharded and geo-replicated data store should offer. On the one hand, causal consistency maintains most of the performance edge of eventual consistency [60] over strong consistency, as all replicas are available for reads under network partitions [30, 38]. On the other hand, it minimizes the associated baggage of increased programmer complexity and user-visible anomalies. By ensuring that all clients see updates that may potentially be causally related [34] in the same order, causal consistency can, for example, address the race conditions that a VP of Engineering at Twitter in a 2013 tech talk called "the biggest problem for Twitter" [33]: when fanning out tweets from celebrities with huge followings, some feeds may receive reactions to the tweets before receiving the tweets themselves.

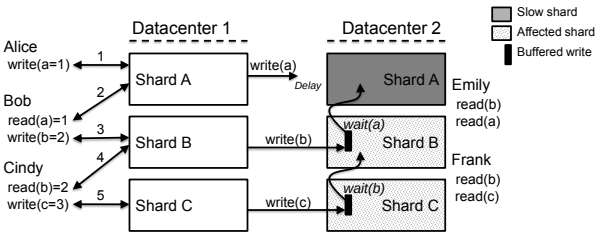


Figure 1: Example of a slowdown cascade in traditional causal consistency. Delayed replicated write(a) delays causally dependent replicated write(b) and write(c)

Despite these obvious benefits, however, causal consistency is largely not deployed in production systems, as existing implementations are liable to experience, at scale, one of the downsides of strong consistency: *slowdown cascades*.

2.1 Slowdown Cascades

When systems scale to sufficient size, failures become an inevitable and regular part of their operation [25, 26]. Performance anomalies, e.g., one node running with lower throughput than the rest of the system, are typical, and can be viewed as a kind of partial failure. Potential causes of such failures include abnormally-high read or write traffic, partially malfunctioning hardware, or a localized network issue, like congestion in a top-of-rack switch. In a partitioned system, a failure within a partition will inevitably affect the performance of that partition. A *slowdown cascade* occurs when the failure spills over to affect other partitions.

Industry has long identified the spectre of slowdown cascades as one of the leading reasons behind its reluctance to build strongly consistent systems [8, 17], pointing out how the slowdown of a single shard, compounded by *query amplification* (e.g., a single user request in Facebook can generate thousands of, possibly dependent, internal queries to many services), can quickly cascade to affect the entire system.

All existing causally consistent systems [14, 28, 39, 40, 63] are susceptible to slowdown cascades. The reason, in essence, is that, to present clients with a causally consistent data store, these systems delay applying a write w until after the data store reflects all the writes that causally precede w. For example, in Eiger [40] each replicated write w carries metadata that explicitly identifies the writes that directly precede w in the causal dependency graph. The datacenter

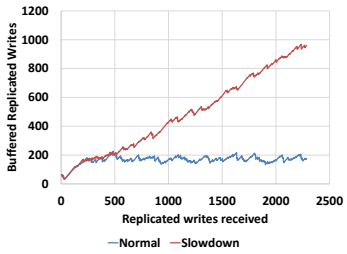


Figure 2: Average queue length of buffered replicated writes in Eiger under normal conditions and when a single shard is delayed by 100 ms.

then delays applying w until these dependencies have been applied locally. The visibility of a write within a shard can then become dependent on the timeliness of other shards in applying their own writes. As Figure 1 shows, this is a recipe for triggering slowdown cascades: because shard A of DC_2 lags behind in applying the write propagating from DC_1, all shards in DC_2 must also wait before they make their writes visible. Shard A's limping inevitably affects Emily's query, but also unnecessarily affects Frank's, which accesses exclusively shards B and C.

In practice, even a modest delay can trigger dangerous slowdown cascades. Figure 2 shows how a single slow shard affects the size of the queues kept by Eiger [40] to buffer replicated writes. Our setup is geo-replicated across two datacenters in Wisconsin and Utah, each running Eiger sharded across 10 physical machines. We run a workload consisting of 95% reads and 5% writes from 10 clients in Wisconsin and a read-only workload from 10 clients in Utah. We measure the average length of the queues buffering replicated writes in Utah. Larger queues mean that newer replicated writes take longer to be applied. If all shards proceed at approximately the same speed, the average queue length remains stable. However, if *any* shard cannot keep up with the arrival rate of replicated writes, then the average queue length across *all* shards grows indefinitely.

3 Observable Causal Consistency

To free causal consistency from slowdown cascades, we revisit what causal consistency *requires*.

Like every consistency guarantee, causal consistency defines a contract between the data store and its users that specifies, for a given set of updates, which values the data store is allowed to return in response to user queries. In particular, causal consistency guarantees that each client observes a monotonically non-decreasing set of updates (including its own), in an order that respects potential causality between operations.

To abide by this contract, existing causally consistent data stores, when replicating writes, enforce internally a stronger invariant than the contract requires: they ensure that clients observe a monotonically non-decreasing set of updates by evolving their data store only through monotonically non-decreasing updates. This strengthening satisfies the contract but, as we saw in Section 2, leaves these systems vulnerable to slowdown cascades.

To resolve this issue, Occult moves the output commit to the clients: letting clients themselves determine when it is safe to read a value frees the data store to make writes visible to clients immediately, without having to first apply all causally preceding writes. Given the duties that many causally consistent data stores already place on their clients (such as maintaining the context of dependencies associated with each of the updates they produce [39, 40]), this is only a small step, but it is sufficient to make Occult impervious to slowdown cascades. Furthermore, Occult no longer needs its clients to be sticky (real-world systems like Facebook sometimes bounce clients between datacenters because of failures, load balancing, and/or load testing [8]). By empowering clients to determine independently whether a read operation is safe, it is no longer problematic to expose a client to the state of a new replica R2 that may not yet reflect some of the updates the client had previously observed on a replica R1.

The general outline of a system that moves the enforcement of causal consistency to read operations is straightforward. Each client c needs to maintain some metadata to encode the most recent state of the data store that it has observed. On reading an object o, c needs to determine whether the version of o that the data store currently holds is safe to read (i.e., if it reflects all the updates encoded in c's metadata): to this end, the data store could keep, together with o, metadata of its own to encode the most recent state known to the client that *created* that version of o. If the version is deemed safe to read, then c needs to update its metadata to reflect any new dependency; if it is not, then c needs to decide how to proceed (among its options: try again; contact a master replica guaranteed to have the latest version of o; or trade safety for availability by accepting a stale version of o).

The key challenge, however, is identifying an encoding of the metadata that minimizes both overhead and read latency. Since each object in the data store must be augmented with this metadata, the importance of reducing its size is obvious; keeping metadata small, however, reduces its ability to track causal dependencies accurately. Any such loss in definition is likely to introduce spurious dependencies between updates. Although these dependencies can never lead to slowdown cascades in Occult, they can increase the chances that read operations will be unnecessarily delayed. Occult's compressed causal timestamps leverage structural and temporal properties to strike a sweet spot between metadata overhead and accuracy (§5).

These causal timestamps have another, perhaps more surprising consequence: they allow Occult to offer the first scalable implementation of causal read-write transactions (§6). Just as the data-store need not be causal, transactions need not take effect atomically in the datastore. They simply need to *appear atomic* to clients. To achieve this, Occult makes

a transaction's writes *causally depend on each other*. This guarantees that clients that seek to read multiple writes from a transaction will independently determine that they must either observe all of the transactions's writes, or none. In contrast, transactions that seek to read a single of the transaction's writes will not be unnecessarily delayed until other replicas have applied writes that they are not interested in. Once again, this is a small step that yields big dividends: transactional writes need no longer be replicated synchronously for safety, obviating the possibility of slowdown cascades.

4 Occult: The Basic Framework

We first outline the system model and an idealized implementation of Occult's basic functionality: clients that read individual objects perceive the data store as causally consistent. We discuss how to make the protocol practical in §5 and sketch Occult's more advanced features (transactions) in §6.

4.1 System Model

Our system is a sharded and replicated key-value store where each replica is located in a separate datacenter with a full copy of the data. The keyspace is divided into a large number of *shards*, i.e., disjoint key ranges. There can be tens or hundreds of thousands of shards, of which multiple can be colocated on the same physical host.

We assume an asynchronous master-slave replication model, with a publicly designated master for every shard. This master shard accepts writes, and asynchronously, but in order, replicates writes to the slave shards. This design is common to several large-scale real-world systems [19, 20, 49, 51] that serve read-heavy workloads with online queries. Master-slave replication has higher write latency than multi-master schemes, but avoids the complexity of dealing with concurrent conflicting writes that can lead to lost updates [39] or require more complex programming models [24].

Clients in our system are co-located with a replica in the same datacenter. Each client reads from its local replica and writes to the master shard (possibly located in a remote replica); a client library enforces causal consistency for reads and attaches metadata to writes. While clients normally read from the shards in their replica, there is no requirement for them to be "sticky" (§3).

4.2 Causal Timestamps

Occult tracks and enforces causal consistency using shardstamps and causal timestamps. A shard's *shardstamp* counts the writes that the shard (master or slave) has accepted. A *causal timestamp* is a vector of shardstamps that identifies a global state across all shards: each entry stores the number of known writes from the corresponding shard. Keeping an entry per shard rather than per object trades-off accuracy against metadata overhead: in exchange for smaller timestamps, it potentially creates false dependencies among all updates to objects mapped to the same shard.

Occult uses causal timestamps for (*i*) encoding the most recent state of the data store observed by a client and (*ii*) capturing the set of causal dependencies for write operations. An object version o created by write w is associated with a causal timestamp that encodes all writes in w's *causal history* (i.e., w and all writes that causally preceded it). Upon reading o, a client updates its causal timestamp to the element-wise maximum of its current value and that of o's causal timestamp: the resulting vector defines the earliest state of the datastore that the client is now allowed to read from to respect causal consistency.

4.3 Basic Protocol

Causal consistency in Occult results from the cooperation between servers and client libraries enabled by causal timestamps. Client libraries use them to validate reads, update them after successful operations, and attach them to writes (Figure 3). Servers store them along with each object, and return one during reads. In addition, servers track the state of each shard using a dedicated shardstamp; when returned in response to a read request, it helps client libraries determine whether completing the read could potentially violate causal consistency (Figure 4).

Write Protocol Occult associates with any value written v a causal timestamp summarizing all of v's causal dependencies. The client library attaches its causal timestamp to every write and sends it to the master of the corresponding shard. The master increments the relevant shardstamp, updates the received causal timestamp accordingly, and stores it with the newly written value. It then asynchronously replicates the write to its slaves, before returning the shardstamp to the client library. Slaves receive writes from the master in order, along with the associated causal timestamps and shardstamps, and update their state accordingly. On receiving the shardstamp, the client library in turn updates *its* causal timestamp to reflect its current knowledge of the shard's state.

Read Protocol A client reads from its local server, which replies with the desired object's most recent value, that value's dependencies (i.e., its causal timestamp), and the current shardstamp of the appropriate shard. The returned shardstamp s makes checking for consistency straightforward. The client simply compares s with the entry of its own causal timestamp for the shard in question (call it s_c) . If s is at least s_c, then the shard already reflects all the local writes that the client has already observed.

When reading from the master shard, the consistency check is guaranteed to succeed. When reading from a slave, however, the check may fail: replication delays from the master shard in another datacenter may prevent a client from observing its own writes at the slave; or the client may have already observed a write in a different shard that *depends* on an update that has not yet reached the slave; .

If the check fails (i.e., the read is *stale*), the client has two choices. It can retry reading from the local replica

```
# cli_ts is the client's causal timestamp

def write(key, value):
  shrd_id = shard(key)
  master_server = master(shrd_id)
  shardstamp = master_server.write(key, value, cli_ts)
  cli_ts[shrd_id] = max(cli_ts[shrd_id], shardstamp)

def read(key):
  shrd_id = shard(key)
  local_server = local(shrd_id)
  value, deps, shardstamp = local_server.read(key)
  cli_ss = cli_ts[shrd_id]
  if isSlave(local_server) and shardstamp < cli_ss:
      return finishStaleRead(key)
  else: cli_ts = entrywise_max(cli_ts, deps)
      return value
```

Figure 3: Client Library Pseudocode

```
1  def write(key, value, deps): #(on masters)
2    shrd_id = shard(key)
3    shardstamps[shrd_id] += 1
4    shardstamp = shardstamps[shrd_id]
5    deps[shard_id] = shardstamp
6    store(key, value, deps)
7    for s in mySlaves(shard_id):
8      async(s.replicate(key, value, deps, shardstamp))
9    return shardstamp
10
11 def replicate(key, value, deps, shardstamp): #(on slaves)
12   shardstamps[shard(key)] = shardstamp
13   storeValue(key, value, deps)
14
15 def read(key):
16   shardstamp = shardstamps[shard(key)]
17   return (getValue(key), getDeps(key), shardstamp)
```

Figure 4: Server Pseudocode

until the shardstamp advances enough to clear the check. Alternatively, it can send the read to the master shard, which always reflects the most recent state of the shard, at the cost of increased latency and additional load on the master. Occult adopts a hybrid strategy: it retries locally for a maximum of r times (with an exponentially increasing delay between retries) and only then reads from the master replica. This approach resolves most stales quickly, while preventing clients from overloading their local slaves with excessive retries.

Finally, the client updates its causal timestamp to reflect the dependencies included in the causal timestamp returned by the server, ensuring that future successful reads will never be inconsistent with the last read value.

5 Causal Timestamp Compression

The above protocol relies on causal timestamps with an entry per shard, a prohibitive proposition when the number of shards N can be in the hundreds of thousands. Occult compresses their size to n entries (with $n \ll N$) without introducing many spurious dependencies.

A first attempt: structural compression Our most straightforward attempt—*structural compression*—maps all shards whose ids are congruent modulo n to the same entry, reducing a causal timestamps' size from N to n at the cost of generating spurious dependencies [58]. The impact of these dependencies on performance (in the form of delayed reads) worsens when shards have widely different shardstamps. Suppose shards i and j map to the same entry s_c and their shardstamps read, respectively, 100 and 1000. A client that writes to j will fail the consistency check when reading from a slave of i until i has received at least 1000 writes. In fact, if i never receives 1000 writes, the client will always failover to reading from i's master shard.

These concerns could be mitigated by requiring master shards to periodically advance their shardstamp and then replicate this advancement to their slaves, independent of the write rate from clients. However, fine-tuning the frequency and magnitude of this synchronization is difficult without explicit coordination between i and j. A better solution is instead to rely on *loosely synchronized shardstamps* based on real, rather than logical, clocks [6]. This guarantees that shardstamps differ by no more than the relative offset between their clocks, independent of the write rate on different master shards.

Finally, to reduce the impact of clock skew on creating false dependencies, the master for shard i can use the causal timestamp ts received from a client on a write operation to more tightly synchronize its shardstamp with those of other shards that the client has recently accessed. Rather than blindly using the current value cl of the physical clock of the server on which it is hosted, i can simply set its shardstamp to be larger than the maximum among (*i*) its current shardstamp; (*ii*) cl; and (*iii*) the highest of the values in ts.

Temporal compression Though using real clocks reduces the chances of generating spurious dependencies, it does not fully address the fundamental limitation of using modulo arithmetic to compress causal timestamps: it is still quite likely that shards with relatively far-apart shardstamps will be mapped to the same entry in the causal timestamp vector.

The next step in our refinement is guided by a simple intuition: recent shardstamps are more likely to generate spurious dependencies than older ones. Thus, rather than mapping a roughly equal number of shards to each of its n entries, *temporal compression* focuses a disproportionate fraction of its ability to accurately resolve dependencies on the shards with the most recent shardstamps. Adapting to our purposes a scheme first devised by Adya and Liskov [6], clients assign an individual entry in their causal timestamp to the $n-1$ shards with the most recent shardstamps they have observed. Each entry also explicitly stores the corresponding shard id. All other shards are mapped to the vector's "catch-all" last entry. One may reasonably fear that conflating all but $n-1$ shards in the same entry will lead, when a client tries to read from one of the conflated shards, to a large number of failed consistency checks—but it need not be so. For a large-enough n, the catch-all entry will naturally reflect updates that were accepted a while ago. Thus, when a client tries to read from a

conflated shard i, it is quite likely that the shardstamp of i will have already exceeded the value stored in the catch-all entry.

To allow causal timestamps to maintain the invariant of explicitly tracking the shards with the $n-1$ highest observed shardstamps, we must slightly revise the client's read and write protocols in Figure 3. The first change involves write operations on a shard currently mapped to the catch-all entry. When the client receives back that shard's current shardstamp, it compares it to those of the $n-1$ shards that its causal timestamp is currently tracking explicitly. The shard with the smallest shardstamp joins the ranks of the conflated and its shardstamp, if it exceeds the current value, becomes the new value of the catch-all entry for the conflated shards. The second change occurs on reads and concerns how the client's causal timestamp is merged with the one returned with the object being read. The shardstamps in either of the two causal timestamps are sorted, and only the shards corresponding to the highest $n-1$ shardstamps are explicitly tracked going forward; the others are conflated, and the new catch-all entry updated to reflect the new dependencies it now includes.

Isolating datacenters With either structural or temporal compression, the effectiveness of loosely synchronized timestamps in curbing spurious dependencies can be significantly affected by another factor: the interplay between the time it takes for updates to replicate across datacenters and the relative skew between the datacenters' clocks. Consider two datacenters, A and B, and assume for simplicity a causal timestamp consisting of a single shardstamp. Clocks within each datacenter are closely synchronized and we can ignore their skew. Say, however, that A's clocks run s ms ahead of those in B, that the average replication delay between datacenters is r ms, and that the average interval between consecutive writes at masters is i ms. Assume now that a client c in A writes to a local master node and updates its causal timestamp with the shardstamp it receives. If c then immediately tries to read from a local slave node, c's shardstamp will be ahead of the slave's by about $(s+r+i)$ ms: until the latter catches up, no value read from it will be deemed safe. For clients in B, meanwhile, the window of inconsistency under the same circumstances would be much shorter: just $(-s+r+i)$ ms, potentially leading to substantially fewer stale reads.

This effect can be significant (§8.2.1). The master write interval i, even with a read-heavy Zipfian workload, is less than 1 ms in our experiments. However, the replication delay r can range from a few tens to over 100 ms and cross datacenter clock skew s can be tens of milliseconds even when using NTP [3] (clock skew between nodes in the same datacenter is often within 0.5-2ms). Thus, if masters are distributed across datacenters, the percentage of stale reads experienced by clients of different datacenters can differ by orders of magnitude.

We solve this problem using distinct causal timestamps for each datacenter. On writes, clients use the returned shardstamp to update the causal timestamp of the datacenter hosting the relevant master shard. On reads, clients update each of their datacenter-specific causal timestamps using the corresponding causal timestamps returned by the server.

Two factors mitigate the additional overhead caused by datacenter-specific causal timestamps. First, the number of causal timestamps does not grow with the number of datacenters, but rather with the number of datacenters with master shards, which can be significantly lower [19]. Second, because clocks within each datacenter are closely synchronized, these causal timestamps need fewer entries to achieve a given target in the percentage of stale reads.

6 Transactions

Many applications can benefit from the ability to read and write multiple objects atomically. To this end, Occult builds on the system described for single-key operations to provide general-purpose read-write transactions. To the best of our knowledge, Occult is the first causal system to support general-purpose transactions while being scalable and resilient to slowdown cascades.

Transactions in Occult run under a new isolation property called Per-Client Snapshot Isolation (PC-PSI), a variant of Parallel Snapshot Isolation (PSI) [55]. PSI is an attractive starting point because it aims to strike a careful balance between the competing concerns of strong guarantees (important for developing applications) and scalable low-latency operations. On the one hand, PSI requires that transactions read from a causally consistent snapshot and precludes concurrent conflicting writes. On the other hand, PSI takes a substantial step towards improving scalability by letting transactions first commit at their local datacenter and subsequently replicate their effects asynchronously to other sites (while preserving causal ordering). In doing so, PSI sidesteps the requirement of a total order on *all* transactions, which is the primary scalability bottleneck of Snapshot Isolation [15] (a popular guarantee in non-distributed systems).

PSI's scalability, however, is ultimately undermined by the constraints its implementation imposes on the order in which transactions are to be replicated, leaving it unnecessarily vulnerable to slowdown cascades. Specifically, PSI totally orders all transactions that commit at a replica, and it requires this order to be respected when the transactions are replicated at other sites [55]. For instance, suppose the transactions in Figure 5 are executed by four different clients on the same replica. Under PSI, they would be totally ordered as $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$. If, when these transactions are applied at a different replica, any of the shards in charge of applying T_2 is slow, the replication of T_3 and T_4 will be delayed, even though neither has a read/write dependency on T_2.

PC-PSI removes these unnecessary constraints. Rather than totally ordering all transactions that were coincidentally located on the same replica, PC-PSI only requires transactions to be replicated in a way that respects both read/write

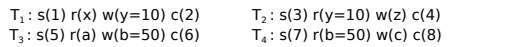


Figure 5: PSI requires transactions to be replicated in commit order. $s(i)$ and $c(j)$ mean respectively start (commit) at timestamp i (j).

dependencies and the order of transactions that belong to the same client session (even when the client is not sticky). This is sufficient to ensure semantically relevant dependencies, i.e., if Alice defriends Bob in one transaction and then later posts her Spring-break photos in another transaction, then Bob will not be able to view her photos, regardless of which replica he reads from. At the same time, it allows Occult to support transactions while minimizing its vulnerability to slowdown cascades.

Like PSI, PC-PSI precludes concurrent conflicting writes. When implementing read-write transactions, this guarantee is crucial to removing the danger of anomalies like lost updates [15]. When writes are accepted at all replicas, as in most existing causally consistent systems [10, 27, 28, 39, 40] this guarantee comes at the cost of expensive synchronization [35], crippling scalability and driving up latency. Not so in Occult, whose master-slave architecture makes it straightforward and inexpensive to enforce, laying the basis for Occult's low-latency read/write transactions.

6.1 PC-PSI Specification

To specify PC-PSI, we start from PSI. In particular, we leverage recent work [23] that proves PSI is equivalent to *lazy consistency* [6]. This isolation level is known [5] to be the weakest to simultaneously provide two guarantees at the core of PC-PSI: (*i*) transactions observe a consistent snapshot of the database and (*ii*) write-write conflicts are not allowed. We thus build on the theoretical framework behind the specification of lazy consistency [5], adding to it the requirement that transactions in the same client session must be totally ordered.

Concretely, we associate with the execution H of a set of transactions a directed serialization graph $DSG(H)$, whose nodes consist of committed transactions and whose edges mark the conflicts (*rw* for read-write, *ww* for write-write, *wr* for write-read) that occur between them. To these, we add a fourth set of edges: $T_i \xrightarrow{sd} T_j$ if some client c first commits T_i and then T_j (*sd* is short for *session dependency*).

The specification of PC-PSI then constrains the set of valid serialization graphs. In particular, a valid $DSG(H)$ must not exhibit any of the following anomalies:

Aborted Reads A committed transaction T_2 reads some object modified by an aborted transaction T_1.

Intermediate Reads A committed transaction T_2 reads a version of an object x written by another transaction T_1 that was not T_1's final modification of x.

Circular Information Flow $DSG(H)$ contains a cycle consisting entirely of *wr*, *ww* and *sd* edges.

Missed Effects $DSG(H)$ contains a cycle that includes exactly one *rw* edge.

Intuitively, preventing Circular Information Flow ensures that if T_1 and T_2 commit and T_1 depends on T_2, then T_2 cannot depend on T_1. In turn, disallowing cycles with a single *rw* edge ensures that no committed transaction ever misses writes of another committed transaction on which it otherwise depends, i.e., committed transactions read from a consistent snapshot and write-write conflicts are prevented (§6.3).

6.2 Executing Read/Write Transactions

Occult supports read/write transactions via a three-phase optimistic concurrency protocol that, in line with the system's ethos, makes clients responsible for running the logic needed to enforce PC-PSI (see Appendix A for the protocol's pseudocode). First, in the *read phase*, a client c executing transaction T obtains from the appropriate shards the objects that T reads, and locally buffers T's writes. Then, in the *validation phase*, c ensures that all read objects belong to a consistent snapshot of the system that reflects the effects of all transactions that causally precede T. Finally, in the *commit phase*, c writes back atomically all objects updated by T.

Read phase For each object o read by T, c contacts the local server for the corresponding shard s_o, making sure, if the server is a slave, not to be reading a stale version (§4.3) of o—i.e., a version of o that is older than what c's causal timestamp already reflects about the state of s_o. If the read is successful, c adds o, its causal timestamp, and s_o's shardstamp to T's *read set*. Otherwise, after a tunable number of further attempts, c proceeds to read o from its master server, whose version is never stale. Meanwhile, all writes are buffered in T's *write set*. They are atomically committed to servers in the final phase. Thus only committed objects are read in this phase and cascading aborts are not possible.

Validation phase Validation involves three steps. In the first, c verifies that the objects in its read set belong to a consistent snapshot Σ_{rs}. It does so by checking that all pairs o_i and o_j of such objects are *pairwise consistent* [12], i.e., that the saved shardstamp of the shard s_{o_i} from which o_i was read is at least as up to date as the entry for s_{o_i} in the causal timestamp of o_j (and vice versa). If the check fails, T aborts.

In the second step, c attempts to lock every object o updated by a write w in T's write set by contacting the corresponding shard s_o on the master server. If c succeeds, Occult's master-slave design ensures that c has exclusive write access to the latest version of o (reads are always allowed); if not, c restarts this step of the validation phase until it succeeds (or possibly aborts T after n failed attempts). In response to a successful lock request, the master server returns two data items: 1) o's causal timestamp, and 2) the new shardstamp that will be assigned to w. c stores this information in T's *overwrite set*. Note that, since they have been obtained from the corresponding master servers, the causal timestamps of the objects in the overwrite set are guaranteed to be pairwise consistent, and therefore to define a consistent snapshot Σ_{ow}: Σ_{ow} captures the updates of all transactions that T would depend on after committing.

To ensure that T is not missing any of these updates, in the final step of validation c checks that Σ_{rs} is at least as recent as Σ_{ow}. If the check fails, T aborts.

Commit phase c computes T's *commit timestamp* ts_T by first initializing it to the causal timestamp of the snapshot Σ_{rs} from which T read, and by then updating it to account for the shardstamps, saved in T's overwrite set, assigned to T's writes. The value of $ts_T[i]$ is thus set to the largest between (*i*) the highest value of the i-th entry of any of the causal timestamps in T's read set, and (*ii*) the highest shardstamp assigned to any of the writes in T's write set that update an object stored on a shard mapped to entry i. c then writes back the objects in T's write set to the appropriate master server, with ts_T as their causal timestamp. Finally, to ensure that any future transaction executed by this client will be (causally) ordered after T, c sets its own causal timestamp to ts_T.

The commit phase enforces a property that is crucial for Occult's scalability: it guarantees that transactions are atomic even though Occult replicates their writes asynchronously. Because the commit timestamp ts_T both reflects all writes that T performs and is used as the causal timestamp of every object that T updates, ts_T makes all of these updates, in effect, causally dependent on one another. As a result, any transaction whose read set includes any object o in T's write set will necessarily either become dependent on all the updates that T performed, or none of them.

6.3 Correctness

To implement PC-PSI, the protocol must prevent Aborted Reads, Intermediate Reads, Circular Information Flow, and Missed Effects. The optimistic nature of the protocol trivially yields the first two conditions, as writes are buffered locally and only written back when transactions commit. Occult also precludes Circular Information Flow. Since clients acquire write locks on all objects before modifying them, transactions that modify the same objects cannot commit concurrently and interleave their writes (no *ww* cycles). Cycles consisting only of *ww*, *wr*, and *sd* edges are instead prevented by the structure of OCC, whose read phase strictly precedes all writes: if a sequence of *ww*/*wr*/*sd* edges leads from T_1 to T_2, then T_1 must have committed before T_2, and could not have observed the effects of T_2 or created a write with a lower causal timestamp than T_2's.

Finally, Occult's validation phase prevents Missed Effects. By contradiction, suppose that all transactions involved in a DSG cycle with a single anti-dependency (*rw*) edge have passed the validation phase. Let T be the transaction from which that edge originates, ending in T^*. Let T_{-1} immediately precede T in the cycle. Let o be the object written by T^* whose update T missed. Either T_{-1} and T^* are one, or T_{-1} *wr*/*ww*/*sd* depends on T^*: either way, Occult's protocol ensures that the commit timestamp of T_{-1} is at least as large as that of T^*. By assumption, T missed some update to o: hence, the shardstamp for o's shard s_o in T's readset must be smaller that the corresponding entry in the commit timestamps of T^* and T_{-1}. There are three cases:

(*i*) $T_{-1} \xrightarrow{sd} T$. The client that issued both T_{-1} and T must have decreased its causal timestamp after committing T_{-1}, but the protocol ensures causal timestamps increase monotonically.

(*ii*) $T_{-1} \xrightarrow{ww} T$. Since T overwrites an object updated by T_{-1}, T's overwrite set must include T_{-1}'s commit timestamp. But then T would fail in validating its read set against its overwrite set, since the latter has a larger entry corresponding to s_o than the former.

(*iii*) $T_{-1} \xrightarrow{wr} T$. Since T reads an object updated by T_{-1}, its read set contains T_{-1}'s commit timestamp. But then T would fail in validating its read set, since the object updated by T_{-1} and the version of o read by T would be pairwise inconsistent.

Each case leads to a contradiction: hence no such cycle can occur and no effects are missed.

7 Fault Tolerance

Server failures Slave failures in Occult only increase read latency as slaves never accept writes and read requests to failed slaves eventually time-out and redirect to the master. Master failures are more critical. First, as in all single-master systems [56], no writes can be processed on a shard with a failed master. Second, in common with all asynchronously replicated systems [11, 14, 39, 40, 56], Occult exhibits a vulnerability window during which writes executed at the master may not yet have been replicated to slaves and may be lost if the master crashes. These missing writes may cause subsequent client requests to fail: if a client c's write to object o is lost, c cannot read o without violating causality. This scenario is common to all causal systems for which clients do not share fate with the servers to which they write. Occult's client-centric approach to causal consistency, however, creates another dangerous scenario: as datacenters are not themselves causally consistent, writes can be replicated out of order. A write y that is dependent on a write x can be replicated to another datacenter despite the loss of x, preventing any subsequent client from reading both x and y.

Master failures can be handled using well-known techniques: individual machine failures within a datacenter can be handled by replicating the master locally using chain-replication [59] or Paxos [36], before replicating asynchronously to other replicas.

Client failures A client failure for single-key operations impacts only the failed client as neither reads nor writes create temporary server state. In transactional mode, however, clients modify server state during the commit phase: they acquire locks on objects in the transaction's write-set and write back new values. A client failure during the transaction commit process may thus cause locks to be held indefinitely by failed clients, preventing other transactions from committing. Such failures can be handled by augmenting Occult with Bernstein's cooperative termination protocol [16] for coor-

dinator recovery [32, 64]. Upon detecting a suspected client failure, individual shards can attempt to elect themselves as backup coordinator (using an instance of Paxos to ensure that a single coordinator is elected). The backup coordinator can then appropriately terminate the transaction (by committing it if a replica shard successfully received an unlock request with the appropriate transaction timestamp using the buffered writes at every replica, or aborting it otherwise).

8 Evaluation

Our evaluation answers three questions:

1. How well does Occult perform in terms of throughput, latency, and transaction abort rate?
2. What is its overhead when compared to an eventually-consistent system?
3. What is the effect of server slowdowns on Occult?

We have implemented Occult by modifying Redis Cluster [4], the distributed implementation of the widely-used Redis key-value store. Redis Cluster divides the entire key-space into N logical shards (default N = 16K), which are then evenly distributed across the available physical servers. Our causal timestamps track shardstamps at the granularity of logical shards to avoid dependencies on the physical location of the data.

For a fair comparison with Occult, we modify our Redis Cluster baseline to allow reads from slaves (Redis Cluster by default uses primary-backup [53] replication for fault tolerance). We further modify the Redis client [2] to, like Occult, allow for client locality: the client prioritizes reading from shards in its local datacenter and executes write operations at the master shard.

8.1 Experimental Setup

Unless otherwise stated, we run our experiments on CloudLab [1, 52] with 20 server and 20 client machines evenly divided across two datacenters in Wisconsin (WI) and South Carolina (SC); the cross-datacenter ping latency is 39ms. Each machine has dual Intel E5-2660 10-core CPUs and dual-port Intel 10Gbe NICs, with respectively 160GB memory (WI) and 256GB (SC). Our experiments use public IP addresses, routable between CloudLab sites, which are limited to 1Gbps. Each server machine runs four instances of the server process, with each server process being responsible for $N/40$ logical shards. Half of all shards have a master in WI and a slave in SC; the other half have the opposite configuration.

Client machines run the Yahoo! Cloud Serving Benchmark (YCSB) [21]. We run experiments with both of YCSB's Zipfian and Uniform workloads but, for brevity, show results only for the Zipfian distribution, more representative of real workloads. Prior to the experiments, we load the cluster with 10 million records following YCSB's default, i.e., keys varying in size up to 23B and 1KB values. We report results at peak goodput, running for at least 100 seconds and then excluding 10-second ramp-up and ramp-down periods. Goodput measures successful operations per second, e.g., a read that needs to be retried four times will only be counted once towards goodput. The bottleneck resource for all experiments is out bound network bandwidth on the hottest master. The CPU on the hottest master is nearly saturated ($>$ 90% utilization) and would almost immediately bottleneck each system at a similar throughput if we were able to increase the network bandwidth.

8.2 Performance and Overhead

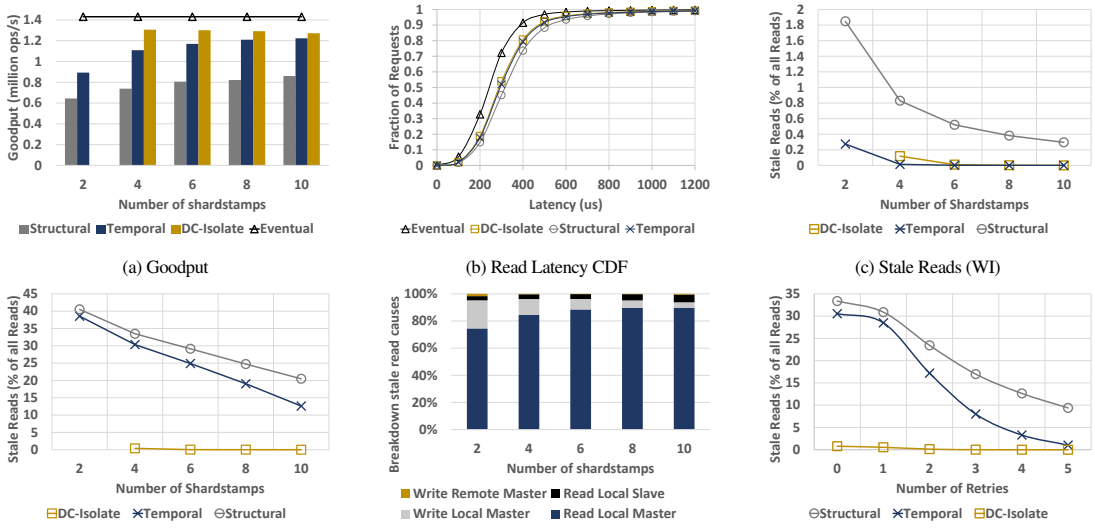
8.2.1 Single Key Operations

We first quantify the overhead of enforcing causal consistency in Occult. We show results for a read-heavy (95% reads, 5% writes) workload, which is more interesting and challenging for our system. Write-heavy workloads performed better in general: we include them in Appendix B.1 for completeness.

We compare system throughput as a function of causal timestamp size, for each of the previously described schemes (structural, temporal, and temporal with datacenter isolation), with Redis cluster as the baseline. Temporal compression requires a minimum of two entries per causal timestamp; adding datacenter isolation (DC-Isolate), doubles this number, so that the smallest number of shardstamps used by DC-Isolate is four.

In the best case (DC-Isolate scheme with four-entry timestamps), Occult's performance is competitive with Redis, despite providing much stronger guarantees: its goodput is only 8.7% lower than Redis (Figure 6a) and its mean and tail latency are, respectively, only 50 µs and 400 µs higher than in Redis (Figure 6b). Other schemes perform either systematically worse (Structural), or require twice the number of shardstamps to achieve comparable performance (Temporal). The low performance of the structural and temporal schemes are due to their high stale read rate (Figures 6c and 6d). In contrast, DC-Isolate has very a low percentage of stale reads even with small causal timestamps. Its slight drop in goodput is primarily due to Occult's other source of overhead: the CPU, network, and storage cost of attaching and storing timestamps to requests and objects. These results highlight the tension between overhead and precision: larger causal timestamps reduce the amount of stale reads (as evidenced by the improved performance of the temporal scheme when vector size grows), but worsen overhead (the goodput of the DC-Isolate scheme actually drops slightly as the number of shardstamps increases).

Achieving a low stale read rate with few shardstamps, as DC-Isolate does, is thus crucial to achieving good performance. Key to its success is its ability to track timestamps from different datacenters independently. Consider Figures 6c and 6d: in these experiments we simply count the percentage of stale reads but do not retry locally or read from the remote master. Observe that the temporal and structural schemes suffer from a significantly higher stale read rate in



(a) Goodput (b) Read Latency CDF (c) Stale Reads (WI)

(d) Stale Reads (SC) (e) Analysis of Stale Read Causes (f) Retrying Stale reads

Figure 6: Measurement and analysis of Occult's overhead for single key operations. Spatial, Temporal or DC-Isolate mean that we run Occult using those compression methods while Eventual indicates our baseline, i.e., Redis Cluster. WI means Wisconsin datacenter and SC means South Carolina datacenter.

the SC datacenter. To understand why, we instrumented the code to track metadata related to the last operation to modify a client's causal timestamp before it does a stale read. We discovered that almost 96% of stale reads occur when the client writes or reads from a local master node immediately before reading from a local slave node (Figure 6e). If the local master node runs ahead (for instance, the SC datacenter has a positive offset of about 22 ms, as measured via *ntpdate*), the temporal scheme will declare all reads to the local slave as stale. In contrast, by tracking dependencies on a per-datacenter basis, DC-Isolate side-steps this issue, producing a low stale rate across datacenters.

8.2.2 Transactions

To evaluate transactions, we modify the workload generator of the YCSB benchmark to issue *start* and *commit* operations in addition to reads and writes. Operations are dispatched serially, i.e., operation i must complete before operation $i+1$ is issued. The resulting long duration of transactions are worst case scenario for Occult. The generator is parameterized with the required number of operations per transaction (T_{size}). We use the DC-Isolate scheme for Occult in all these experiments.

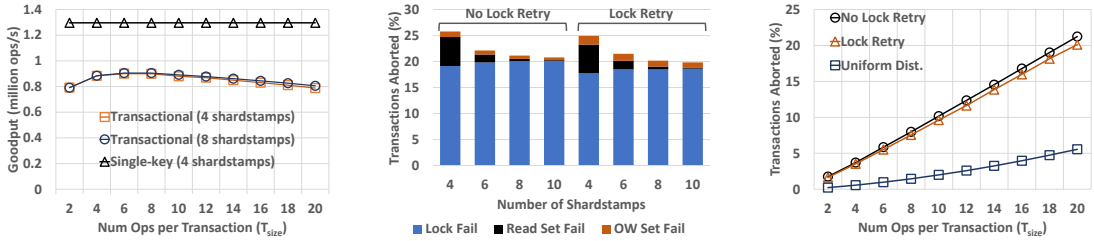
We show results for increasing values of T_{size}. For smaller values, most transactions in the workload are read-only, and as T_{size} increases most transactions become read-write . As Figure 7a shows, the overall goodput remains within 2/3 of the goodput of non transactional Occult (varying from 60% to 70%), even as T_{size} increases and aborts become more likely. Figures 7b and 7c analyze the causes of these aborts. Recall from §6.2 that aborts can occur because of either (*i*) validation failures of the read/overwrite sets or (*ii*) failure to acquire locks on keys being written. Figure 7b fixes $T_{size}=20$ and classifies aborts into these three categories. We find that aborts are dominated by the failure to acquire locks. Furthermore, due to the highly skewed nature of the YCSB zipfian workload, >80% of these lock-fail aborts are due to contention on the 50 hottest keys. This high contention also explains the limited benefit of retrying to acquire locks. Figure 7b also shows that increasing the number of shardstamps almost completely eliminates aborts due to failed validations of the read set and roughly halves aborts due to failed validations of the overwrite set. Finally, in Figure 7c retrying lock acquisition has slightly better impact at larger values of T_{size} when most transactions are read-write. For comparison, we show the abort rate on a uniform distribution.

8.2.3 Resource Overhead

To quantify the resource overhead of Occult over Redis Cluster, we measure the CPU usage (using *getrusage*()) and the total bytes sent and received over 120 secs for both systems at the same throughput (1.27Mop/s) and report the average of five runs, averaged over the 80 server processes.

Overall CPU usage increases by 7% with a slightly higher increase on slaves (8%) than masters (6%). This difference is due to stale read retries in Occult. Output bandwidth increases by 8.8%, while input bandwidth increases by 49%, as attaching metadata to read requests with a key size of at most 23B has a much larger impact than attaching it to replies carrying 1KB values.

Finally, we measure storage overhead by loading both Redis and Occult with 10 million records and measuring the increase in memory usage of each server process. Storing four shardstamps with each key results in an increase, on average, of 3% for Occult over Redis. Storing 10 shardstamps instead results in an increase of 4.9%.



(a) Goodput

(b) Abort Rate as $f(shardstamps)$ ($T_{size}=20$)

(c) Abort Rate as $f(T_{size})$ ($shardstamps=8$)

Figure 7: Transactions in Occult

8.3 Impact of slow nodes

Occult is by design immune to the slowdown of a server cascading to affect the entire system. Nonetheless, the slowdown of a server does introduce additional overhead relative to an eventually-consistent system. In particular, slowing down slaves increases the stale rate, which in turn increases retries on that slave and remote reads from its corresponding master. We measure these effects by artificially slowing down the replication of writes at a number of slave nodes, symmetrically increasing their number from one to three per datacenter—two to six overall. This causes a slowdown of around 2.5% to 7.5% of all nodes. We notice that, at peak throughput, the node containing the hottest key serves around 3× more operations than the nodes serving keys in the tail of the distribution. We evaluate slowdowns of the tail nodes separately from the hot nodes, which we slowdown in decreasing order of load, starting from the hottest node.

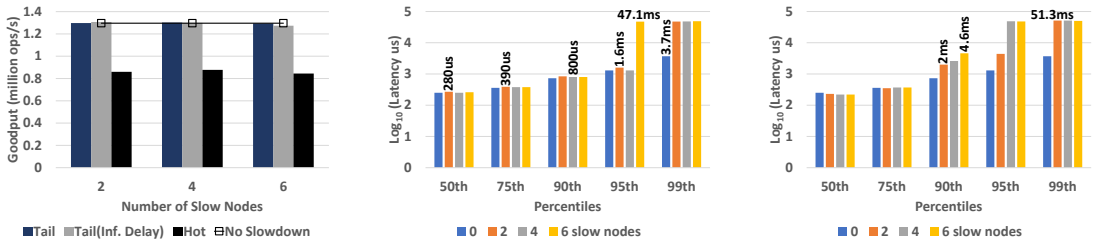
We first delay replicated writes on tail nodes by 100 ms, which, as Figure 8a shows, does not affect throughput: even at peak throughput for the cluster, only the hottest nodes are actually CPU or network saturated. As such, tail nodes (master or slave) still have spare capacity. When clients failover to the master (after n local retries), this spare capacity absorbs the additional load. In contrast, read latency is affected (Fig 8b). Though median, 75th, and 90th percentile latencies remain unchanged because reads to non-slow nodes are unaffected by the presence of slow servers, tail latencies increase significantly as the likelihood of hitting a lagging server and reading a stale value increases. Thus, increasing slow nodes from two to six first makes the 99th percentile and then the 95th percentile latency jump to around 48ms. This includes $n=4$ local retries by the client (after delays of 0, 1, 2, and 4 ms) and finally contacting the master in a remote datacenter (39 ms away). Having a large delay of 100 ms and $n=4$ means that our experiment actually evaluates an arbitrarily large slowdown, since almost all client reads to slow slaves eventually fail over to the master. We confirm this by setting the delay to infinite: the results for both throughput (Figure 8a) and latency (not shown) are identical to the 100 ms case.

Slowing down the hot nodes impacts both throughput and latency. The YCSB workload we use completely saturates the hottest master and its slave. Unlike in the previous experiments, the hot master does not have any spare capacity to handle failovers, and throughput suffers (Figure 8a). Slowing more than two slave nodes does not decrease throughput further becuase their respective masters have spare capacity. Figure 8c shows that, as expected given the skewed workload, slowing down an increasing number of hot nodes increases the 99th and 95th percentile latencies faster than slowing down tail nodes (Figure 8b). The median and 75th percentile latencies remain unchanged as before.

9 Related Work

Scalable Causal Consistency COPS [39] tracks causal consistency with explicit dependencies and then enforces it pessimistically by checking these dependencies before applying remote writes at a replica. COPS strives to limit the loss of throughput caused by the metadata and messages needed to check dependencies by exploiting transitivity. ChainReaction [10], Orbe [27], and GentleRain [28] show how to reduce these dependencies further by using Bloom filters, dependency matrices, and a single timestamp, respectively. These techniques reduce metadata by making it more coarse-grained, which actually exacerbates slowdown cascades. Eiger [40] builds on COPS with a more general data model, write-only transactions, and an improved read-only transaction algorithm. BoltOn [14] shows how to use shim layers to add pessimistic causal consistency to an existing eventually consistent storage system. COPS-SNOW [41] provides a new latency-optimal read-only transaction algorithm. Occult improves on this line of research by identifying the problem of slowdown cascades and showing how an optimistic approach to causal consistency can overcome them. In addition, all of these systems provide weaker forms of transactions than Occult: Eiger provides read-only and write-only transactions, while all other systems provide only read-only transactions or no transactions at all.

Pileus [56] and Tuba [11] (which adds reconfigurability to Pileus) provide a range of consistency models that clients can dynamically choose between by specifying an SLA that assigns utilities to different combinations of consistency and latency. Pileus has several design choices that are similar to Occult: it uses a single master, applies writes at replicas without delay (i.e., is optimistic), uses a timestamp to determine



(a) Goodput (b) Latency(slow tail nodes) (c) Latency(slow hot nodes)

Figure 8: Effect on overall goodput and read latency due to slow nodes in Occult

if a read value meets a given consistency level (including causal consistency), and can issue reads across different datacenters to meet a given consistency level. However, Pileus is not scalable as it uses a single logical timestamp as the client's state (which we show in our evaluation has a very high false positive stale rate) and evaluates with only a single node per replica. We consider an interesting avenue of future work to see if we can combine the focus of Pileus (consistency choice and SLAs) with Occult.

Cure [9] is a causally consistent storage system that provides read-write transactions. Cure is pessimistic and uses a single timestamp per replica to track and enforce causal dependencies. Cure provides a restricted form of read-write transactions that requires all operations to be on convergent and commutative replicated data types (CRDTs) [54]. Using CRDTs allows Cure to avoid coordination for writes and instead eventually merges conflicting writes, including those issued as part of read-write transactions. Occult, in contrast, is an optimistic system that provides read-write transactions for the normal data types that programmers are familiar with. Saturn [18], like Occult, tries to strike a balance between metadata overhead and false sharing by relying on "small labels" (like Cure) while selecting serializations at datacenters that minimize spurious dependencies.

Read/Write Transactions Many recent research systems with read/write transactions are limited to a single datacenter (e.g., [37, 46, 61, 62]) whereas most production systems are geo-replicated. Some geo-replicated research systems cannot scale to large clusters because they have a single point of serialization per datacenter [24, 55] while others are limited to transactions with known read and write sets [47, 57, 65].

Scalable geo-replicated transactional systems include Spanner [22], MDCC [32], and TAPIR [64]. Spanner is a production system at Google that uses synchronized clocks to reduce coordination for strictly serializable transactions. MDCC uses Generalized Paxos [36] to reduce wide-area commit latency. TAPIR avoids coordination in both replication and concurrency control to be able to sometimes commit a transaction in a single wide-area round trip. All of these systems provide strict serializability, a much stronger consistency level than what Occult provides. As a result, they require heavier-weight mechanisms for deciding to abort or commit transactions and will abort more often.

Rethinking the Output Commit Step We were inspired to rethink the output commit step for causal consistency by a number of previous systems: Rethink the Sync [48], which did it for local file I/O; Blizzard [45], which did it for cloud storage; Zyzzyva [31], which did it for Byzantine fault tolerance; and Speculative Paxos [50], which did it for Paxos.

10 Conclusion

This paper identifies slowdown cascades as a fundamental limitation of enforcing causal consistency as a global property of the datastore. Occult instead moves this responsibility to the client: the data store makes its updates available as soon as it receives them. Clients then enforce causal consistency on reads only for updates that they are actually interested in observing, using compressed timestamps to track causality. Occult follows the same philosophy for its scalable general-purpose transaction protocol: by ensuring that transactions read from a consistent snapshot and using timestamps to guarantee atomicity, it guarantees the strong properties of PSI while avoiding its scalability bottleneck.

11 Acknowledgements

We are grateful to our shepherd Jay Lorch for his dedication to making this paper as good as it could be, and to the anonymous reviewers for their insightful comments. This work would simply not have been possible without the patience and support of the amazing CloudLab team [1] throughout our experimental evaluation. This work was supported by the National Science Foundation under grant number CNS-1409555 and by a Google Faculty Research Award. Natacha Crooks was partially supported by a Google Doctoral Fellowship in Distributed Computing.

References

[1] CloudLab. https://www.cloudlab.us/.

[2] Jedis. https://github.com/xetorthio/jedis.

[3] Network Time Protocol. https://www.eecis.udel.edu/~mills/ntp.html.

[4] Redis Cluster Specification. http://redis.io/topics/cluster-spec.

[5] ADYA, A. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, MIT, 1999.

[6] ADYA, A., AND LISKOV, B. Lazy Consistency Using Loosely Synchronized Clocks. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing* (Santa Barbara, California, USA, 1997), PODC '97, ACM, pp. 73–82.

[7] AHAMAD, M., NEIGER, G., BURNS, J., KOHLI, P., AND HUTTO, P. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing 9*, 1 (1995), 37–49.

[8] AJOUX, P., BRONSON, N., KUMAR, S., LLOYD, W., AND VEERARAGHAVAN, K. Challenges to Adopting Stronger Consistency at Scale. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Switzerland, 2015), HOTOS'15, USENIX Association.

[9] AKKOORATH, D. D., TOMSIC, A. Z., BRAVO, M., LI, Z., CRAIN, T., BIENIUSA, A., PREGUIA, N., AND SHAPIRO, M. Cure: Strong Semantics Meets High Availability and Low Latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)* (June 2016), pp. 405–414.

[10] ALMEIDA, S., LEITÃO, J. A., AND RODRIGUES, L. Chainreaction: A Causal+ Consistent Datastore Based on Chain Replication. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic, 2013), EuroSys '13, ACM, pp. 85–98.

[11] ARDEKANI, M. S., AND TERRY, D. B. A Self-Configurable Geo-Replicated Cloud Storage System. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), OSDI '14, USENIX Association, pp. 367–381.

[12] BABAOĞLU, O., AND MARZULLO, K. Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. In *Distributed Systems (2nd Ed.)*, S. Mullender, Ed. ACM Press/Addison-Wesley Publishing Co., 1993, pp. 55–96.

[13] BAILIS, P., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. The Potential Dangers of Causal Consistency and an Explicit Solution. In *Proceedings of the 3rd ACM Symposium on Cloud Computing* (San Jose, California, 2012), SoCC '12, ACM, pp. 22:1–22:7.

[14] BAILIS, P., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Bolt-On Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, 2013), SIGMOD '13, ACM, pp. 761–772.

[15] BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O'NEIL, E., AND O'NEIL, P. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (San Jose, California, USA, 1995), SIGMOD '95, ACM, pp. 1–10.

[16] BERNSTEIN, P., AND NEWCOMER, E. *Principles of Transaction Processing: For the Systems Professional*. Morgan Kaufmann Publishers Inc., 1997.

[17] BIRMAN, K., CHOCKLER, G., AND VAN RENESSE, R. Toward a Cloud Computing Research Agenda. *SIGACT News 40*, 2 (June 2009), 68–80.

[18] BRAVO, M., RODRIGUES, L., AND VAN ROY, P. Saturn: a distributed metadata service for causal consistency. In *Proceedings of the 12th ACM European Conference on Computer Systems* (2017), EuroSys '17, ACM.

[19] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., MARCHUKOV, M., PETROV, D., PUZAR, L., SONG, Y. J., AND VENKATARAMANI, V. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference* (San Jose, CA, 2013), USENIX ATC'13, USENIX Association, pp. 49–60.

[20] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proceedings of the VLDB Endowment 1*, 2 (Aug. 2008), 1277–1288.

[21] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, 2010), SoCC '10, ACM, pp. 143–154.

[22] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's Globally-Distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, 2012), OSDI'12, USENIX Association, pp. 251–264.

[23] CROOKS, N., PU, Y., ALVISI, L., AND CLEMENT, A. Seeing is Believing: A Unified Model for Consistency and Isolation via States. *CoRR abs/1609.06670* (2016).

[24] CROOKS, N., PU, Y., ESTRADA, N., GUPTA, T., ALVISI, L., AND CLEMENT, A. TARDiS: A Branch-and-Merge Approach to Weak Consistency. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data* (San Francisco, California, 2016), SIGMOD '16, ACM, pp. 1615–1628.

[25] DEAN, J., AND BARROSO, L. A. The Tail at Scale. *Communications of the ACM 56*, 2 (Feb. 2013), 74–80.

[26] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, 2004), OSDI'04, USENIX Association, pp. 137–149.

[27] DU, J., ELNIKETY, S., ROY, A., AND ZWAENEPOEL, W. Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In *Proceedings of the 4th ACM Symposium on Cloud Computing* (Santa Clara, California, 2013), SOCC '13, ACM, pp. 11:1–11:14.

[28] DU, J., IORGULESCU, C., ROY, A., AND ZWAENEPOEL, W. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In *Proceedings of the 5th ACM Symposium on Cloud Computing* (2014), SOCC '14, ACM.

[29] FIDGE, C. J. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. In *Proceedings of the 11th Australian Computer Science Conference (ACSC'88)* (February 1988), pp. 56–66.

[30] GILBERT, S., AND LYNCH, N. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News 33*, 2 (June 2002), 51–59.

[31] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems 27*, 4 (Jan. 2010), 7:1–7:39.

[32] KRASKA, T., PANG, G., FRANKLIN, M. J., MADDEN, S., AND FEKETE, A. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic, 2013), EuroSys '13, ACM, pp. 113–126.

[33] KRIKORIAN, R. Twitter Timelines at Scale (video link. consistency discussion at 26m). http://www.infoq.com/presentations/Twitter-Timeline-Scalability, 2013.

[34] LAMPORT, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM 21*, 7 (July 1978), 558–565.

[35] LAMPORT, L. The Part-Time Parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.

[36] LAMPORT, L. Generalized Consensus and Paxos. Tech. Rep. MSR-TR-2005-33, Microsoft Research, 2004.

[37] LEE, C., PARK, S. J., KEJRIWAL, A., MATSUSHITA, S., AND OUSTERHOUT, J. Implementing Linearizability at Large Scale and Low Latency. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California, 2015), SOSP '15, ACM, pp. 71–86.

[38] LIPTON, R. J., AND SANDBERG, J. PRAM: A Scalable Shared Memory. Tech. Rep. TR-180-88, Princeton University, Department of Computer Science, August 1988.

[39] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Cascais, Portugal, 2011), SOSP '11, ACM, pp. 401–416.

[40] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Lombard, IL, 2013), NSDI '13, USENIX Association, pp. 313–328.

[41] LU, H., HODSDON, C., NGO, K., MU, S., AND LLOYD, W. The SNOW Theorem and Latency-Optimal Read-Only Transactions. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (2016), OSDI'16, USENIX Association, pp. 135–150.

[42] LU, H., VEERARAGHAVAN, K., AJOUX, P., HUNT, J., SONG, Y. J., TOBAGUS, W., KUMAR, S., AND LLOYD, W. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California, 2015), SOSP '15, ACM, pp. 295–310.

[43] MAHAJAN, P., ALVISI, L., AND DAHLIN, M. Consistency, Availability, and Convergence. Tech. Rep. UTCS TR-11-22, Department of Computer Science, The University of Texas at Austin, 2011.

[44] MATTERN, F. Virtual Time and Global States of Distributed Systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms* (1989), North-Holland/Elsevier, pp. 215–226.

[45] MICKENS, J., NIGHTINGALE, E. B., ELSON, J., GEHRING, D., FAN, B., KADAV, A., CHIDAMBARAM, V., KHAN, O., AND NAREDDY, K. Blizzard: Fast, Cloud-Scale Block Storage for Cloud-Oblivious Applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), USENIX Association, pp. 257–273.

[46] MU, S., CUI, Y., ZHANG, Y., LLOYD, W., AND LI, J. Extracting More Concurrency from Distributed Transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO, 2014), OSDI'14, USENIX Association, pp. 479–494.

[47] MU, S., NELSON, L., LLOYD, W., AND LI, J. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (2016), OSDI'16, USENIX Association.

[48] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the Sync. *ACM Transactions on Computer Systems 26*, 3 (Sept. 2008), 6:1–6:26.

[49] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Lombard, IL, 2013), NSDI '13, USENIX Association, pp. 385–398.

[50] PORTS, D. R., LI, J., LIU, V., SHARMA, N. K., AND KRISHNAMURTHY, A. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (2015), USENIX Association, pp. 43–57.

[51] QIAO, L., SURLAKER, K., DAS, S., QUIGGLE, T., SCHULMAN, B., GHOSH, B., CURTIS, A., SEELIGER, O., ZHANG, Z., AURADAR, A., BEAVER, C., BRANDT, G., GANDHI, M., GOPALAKRISHNA, K., IP, W., JGADISH, S., LU, S., PACHEV, A., RAMESH, A., SEBASTIAN, A., SHANBHAG, R., SUBRAMANIAM, S., SUN, Y., TOPIWALA, S., TRAN, C., WESTERMAN, J., AND ZHANG, D. On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, 2013), SIGMOD '13, ACM, pp. 1135–1146.

[52] RICCI, R., EIDE, E., AND THE CLOUDLAB TEAM. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *USENIX ;login: 39*, 6 (Dec. 2014).

[53] SCHNEIDER, F. B. Replication Management Using the State-Machine Approach. In *Distributed Systems (2nd Ed.)*, S. Mullender, Ed. ACM Press/Addison-Wesley Publishing Co., 1993, pp. 169–197.

[54] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. A Comprehensive Study of Convergent and Commutative Replicated Data Types. Tech. Rep. HAL Id: inria-00555588, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[55] SOVRAN, Y., POWER, R., AGUILERA, M. K., AND LI, J. Transactional Storage for Geo-Replicated Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Cascais, Portugal, 2011), SOSP '11, ACM, pp. 385–400.

[56] TERRY, D. B., PRABHAKARAN, V., KOTLA, R., BALAKRISHNAN, M., AGUILERA, M. K., AND ABU-LIBDEH, H. Consistency-based Service Level Agreements for Cloud Storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (Farmington, Pennsylvania, 2013), SOSP '13, ACM, pp. 309–324.

[57] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), ACM, pp. 1–12.

[58] TORRES-ROJAS, F. J., AND AHAMAD, M. Plausible clocks: Constant size logical clocks for distributed systems. *Distributed Computing 12*, 4 (Sept. 1999), 179–195.

[59] VAN RENESSE, R., AND SCHNEIDER, F. B. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, 2004), OSDI'04, USENIX Association, pp. 91–104.

[60] VOGELS, W. Eventually Consistent. *Commun. ACM 52*, 1 (Jan. 2009), 40–44.

[61] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast In-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California, 2015), SOSP '15, ACM, pp. 87–104.

[62] XIE, C., SU, C., LITTLEY, C., ALVISI, L., KAPRITSOS, M., AND WANG, Y. High-Performance ACID via Modular Concurrency Control. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California, 2015), SOSP '15, ACM, pp. 279–294.

[63] ZAWIRSKI, M., PREGUIÇA, N., DUARTE, S., BIENIUSA, A., BALEGAS, V., AND SHAPIRO, M. Write Fast, Read in the Past: Causal Consistency for Client-Side Applications. In *Proceedings of the 16th Annual Middleware Conference* (Vancouver, BC, Canada, 2015), Middleware '15, ACM, pp. 75–87.

[64] ZHANG, I., SHARMA, N. K., SZEKERES, A., KRISHNAMURTHY, A., AND PORTS, D. R. K. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California, 2015), SOSP '15, ACM, pp. 263–278.

[65] ZHANG, Y., POWER, R., ZHOU, S., SOVRAN, Y., AGUILERA, M. K., AND LI, J. Transaction Chains: Achieving Serializability with Low Latency in Geo-Distributed Storage Systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (Farmington, Pennsylvania, 2013), SOSP '13, ACM, pp. 276–291.

A Pseudocode for Transactions

Listing 1: Interface of a Causal Timestamp

```
class CausalTimestamp:
  def init(N):
    V = [0] * N

# Get shardstamp for shard_id
  def getSS(shard_id):
    return V[shard_id]

# Return the shardstamp with maximum value
  def maxSS():
    return max(V)

# Update the shardstamp for shard_id to new_ss
  def updateSS(shard_id,new_ss):
    V[shard_id] = max(V[shard_id], new_ss)

# Merge another CausalTimestamp into this object
 def mergeCTS(other_cts):
    for i in range(0, len(V)):
      V[i] = max(V[i], other_cts[i])
```

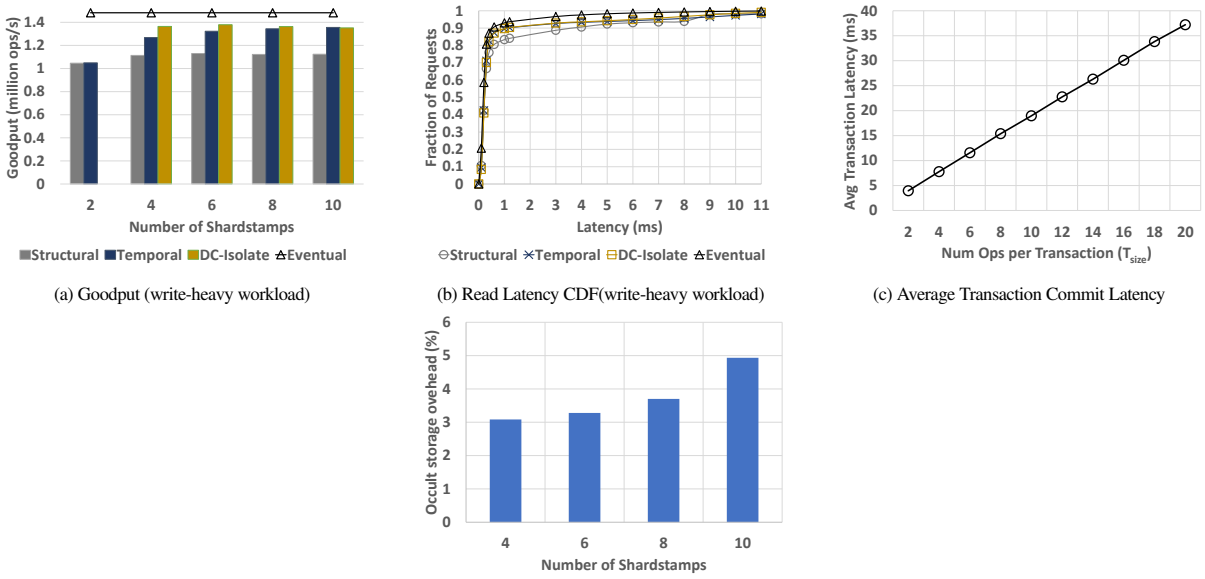
Listing 2: Server-side pseudocode

```
# allocate new shardstamp using loosely synchronized
# clocks as described in Section 5
def newShardstamp(max_cli_ss, shard_id):
  new_ss = max(currentSysTime(), max_cli_ss)
  if new_ss < shardstamps[shard_id]:
    return shardstamps[shard_id] + 1
  else:
    return new_ss + 1

def read(key):
  shardstamp = shardstamps[shard(key)]
  return (getValue(key),getDeps(key),shardstamp)

def prepare(tid, key, value, max_cli_ss):
  if not islocked(shard(key)):
    lockwrites(shard(key))
    if tid not in prepKV: # prepared txns key vals
      prepKV[tid] = list()
    prepKV[tid].append((key, value))
    shardstamp = shardstamps[shard(key)]
    new_ss = newShardstamp(max_cli_ss, shard(key))
    return (new_ss, getDeps(key))
  else:
    throw LOCKED

def commit_server(tid, deps):
  for key, value in prepKV[tid]:
    shardstamps[shard(key)] = deps.maxSS()
    store(key, value, deps)
    shardstamp = shardstamps[shard(key)])
    unlockwrites(shard(key))
    for s in mySlaves():
      async(s.replicate(key,value,deps,shardstamp)

def abort_server(tid):
  for key, value in prepKV[tid]:
    unlockwrites(key)
```

Note that if multiple transactions concurrently update different objects in the same shard *s*, in the commit phase each write *w* is applied at *s* (and at its slaves) in the (total) order determined by the value of the shardstamp assigned to *w* during the validation phase. The pseudocode achieves this property by locking shards instead of objects during the validation phase (§6.2).

Listing 3: Client-side pseudocode

```
# cli_ts is the client's causal timestamp
def startTransaction():
  TID = newTransactionID()
  ReadSet = set()
  OWSet = set() # Overwrite Set
  Writes = dict() # Writes done by this transaction
  cli_ts_save = copy(cli_ts)

def write(key, value):
  Writes[key] = value

def read(key):
  if key in Writes:
    return Writes[key] # Return the value we wrote
  else:
    shard_id = shard(key)
    local_server = local(shard_id)
    cli_ss = cli_ts.getSS(shard_id)
    value,deps,shardstamp = local_server.read(key)
    if isSlave(local_server) and shardstamp < cli_ss:
      value,deps,shardstamp = finishStaleRead(key)

    ReadSet.add(Elem(key,shard_id,deps,shardstamp))
    cli_ts.mergeCTS(deps)
    return value

def validate(S1, S2):
  for x in S1:
    for y in S2:
      if x.shardstamp < y.deps.getSS(x.shard_id):
        return False
  return True

def abortTransaction(prepared_servers, tid):
  cli_ts = cli_ts_save
  for server in prepared_servers:
    server.abort_server(tid)
  return False

def commitTransaction():
  prepared_servers = set()
  if not validate(ReadSet, ReadSet):
    return abortTransaction(prepared_servers, TID)

  for key, value in Writes:
    master_server = master(shard(key))
    try:
      max_ss = cli_ts.maxSS()
      new_ss,deps =
          master_server.prepare(TID,key,value,max_ss)
      cli_ts.updateSS(shard(key), new_ss)
      OWSet.add(Elem(key,shard(key),deps))
      prepared_servers.add(master_server)
    except LOCKED: #can retry lock here before abort
      return abortTransaction(prepared_servers, TID)

  if not validate(ReadSet, OWSet):
    return abortTransaction(prepared_servers, TID)
  else:
    for server in prepared_servers:
      server.commit_server(TID, cli_ts)
    return True
```



(a) Goodput (write-heavy workload)

(b) Read Latency CDF(write-heavy workload)

(c) Average Transaction Commit Latency

(d) Storage overhead of Occult over Redis Cluster

Figure 9: Miscellaneous additional evaluation. Spatial, Temporal or DC-Isolate mean that we run Occult using those compression methods while Eventual indicates our baseline, i.e., Redis Cluster.

B Additional Evaluation

In this section we show additional evaluation that could not be shown in the main paper due to space constraints. The experimental setup for this section is identical to the setup from section 8.1 of the main paper.

B.1 Performance on a write-heavy workload

Figures 9a and 9b show evaluation of non-transactional Occult on a write-heavy workload (75% reads, 25% writes) with a Zipfian distribution of operations. Overall Occult suffers less goodput overhead (6.9%) over Redis on this workload than the read-heavy workload. The median latency increase over Redis is still 50µs but tail latency increases by 4ms.

B.2 Commit Latency of Transactions

Figure 9c shows the average commit latency of transactions from *start* to *commit* as a function of T_{size}, i.e., the number of operations in each transaction. The linear rise in latency is because operations in our workload are dispatched serially as discussed in the evaluation of transactions in §8.2.

B.3 Storage overhead of Occult over Redis Cluster

Figure 9d shows the storage overhead of Occult over Redis Cluster with increasing number of shardstamps per causal timestamp. For this experiment, we loaded either system with 10 million records and measured the increase in memory usage of the server processes in Occult. The increase happens since Occult stores causal timestamps with each key.

CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics

Omid Alipourfard[1], Hongqiang Harry Liu[2], Jianshu Chen[2], Shivaram Venkataraman[3], Minlan Yu[1], Ming Zhang[4]

[1] *Yale University,* [2] *Microsoft Research,* [3] *University of California, Berkeley,* [4] *Alibaba Group*

Abstract – Picking the right cloud configuration for recurring big data analytics jobs running in clouds is hard, because there can be tens of possible VM instance types and even more cluster sizes to pick from. Choosing poorly can significantly degrade performance and increase the cost to run a job by 2-3x on average, and as much as 12x in the worst-case. However, it is challenging to automatically identify the best configuration for a broad spectrum of applications and cloud configurations with low search cost. *CherryPick* is a system that leverages Bayesian Optimization to build performance models for various applications, and the models are just accurate enough to distinguish the best or close-to-the-best configuration from the rest with only a few test runs. Our experiments on five analytic applications in AWS EC2 show that *CherryPick* has a 45-90% chance to find optimal configurations, otherwise near-optimal, saving up to 75% search cost compared to existing solutions.

1 Introduction

Big data analytics running on clouds are growing rapidly and have become critical for almost every industry. To support a wide variety of use cases, a number of evolving techniques are used for data processing, such as MapReduce, SQL-like languages, Deep Learning, and in-memory analytics. The execution environments of such big data analytic applications are structurally similar: a cluster of virtual machines (VMs). However, since different analytic jobs have diverse behaviors and resource requirements (CPU, memory, disk, network), their *cloud configurations* – the types of VM instances and the numbers of VMs – cannot simply be unified.

Choosing the right cloud configuration for an application is essential to service quality and commercial competitiveness. For instance, a bad cloud configuration can result in up to 12 times more cost for the same performance target. The saving from a proper cloud configuration is even more significant for *recurring* jobs [10, 17] in which similar workloads are executed repeatedly. Nonetheless, selecting the best cloud configuration, e.g., the cheapest or the fastest, is difficult due to the complexity of simultaneously achieving high accuracy, low overhead, and adaptivity for different applications and workloads.

Accuracy The running time and cost of an application have complex relations to the resources of the cloud instances, the input workload, internal workflows, and configuration of the application. It is difficult to use straightforward methods to model such relations. Moreover, cloud dynamics such as network congestions and stragglers introduce substantial noise [23, 39].

Overhead Brute-force search for the best cloud configuration is expensive. Developers for analytic applications often face a wide range of cloud configuration choices. For example, Amazon EC2 and Microsoft Azure offer over 40 VM instance types with a variety of CPU, memory, disk, and network options. Google provides 18 types and also allows customizing VMs' memory and the number of CPU cores [2]. Additionally, developers also need to choose the right cluster size.

Adaptivity Big data applications have diverse internal architectures and dependencies within their data processing pipelines. Manually learning to build the internal structures of individual applications' performance model is not scalable.

Existing solutions do not fully address all of the preceding challenges. For example, Ernest [37] trains a performance model for machine learning applications with a small number of samples but since its performance model is tightly bound to the particular structure of machine learning jobs, it does not work well for applications such as SQL queries (poor adaptivity). Further, Ernest can only select VM sizes within a given instance family, and performance models need to be retrained for each instance family.

In this paper, we present *CherryPick*—a system that unearths the optimal or near-optimal cloud configurations that minimize cloud usage cost, guarantee application performance and limit the search overhead for recurring big data analytic jobs. Each configuration is represented as the number of VMs, CPU count, CPU speed per core, RAM per core, disk count, disk speed, and network capacity of the VM.

The key idea of *CherryPick* is to build a performance model that is *just accurate enough* to allow us to distinguish near-optimal configurations from the rest. Tolerating the inaccuracy of the model enables us to achieve both low overhead and adaptivity: only a few samples

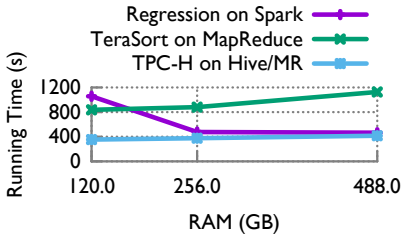


Figure 1: Regression and TeraSort with varying RAM size (64 cores)

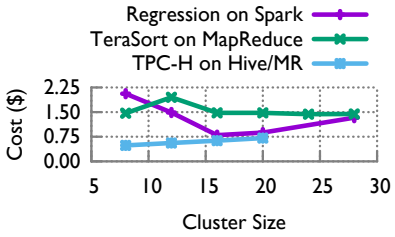


Figure 2: Regression and TeraSort cost with varying cluster size (M4)

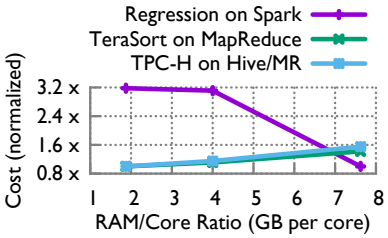


Figure 3: Regression and TeraSort cost with varying VM type (32 cores)

are needed and there is no need to embed application-specific insights into the modeling.

CherryPick leverages Bayesian Optimization (BO) [13, 28, 33], a method for optimizing black-box functions. Since it is non-parametric, it does not have any pre-defined format for the performance model. BO estimates a *confidence interval* (the range that the actual value should fall in with high probability) of the cost and running time under each candidate cloud configuration. The confidence interval is improved (narrowed) as more samples become available. *CherryPick* can judge which cloud configuration should be sampled next to best reduce the current uncertainty in modeling and get closer to go the optimal. *CherryPick* uses the confidence interval to decide when to stop the search. Section 3 provides more details on how BO works and why we chose BO out of other alternatives.

To integrate BO in *CherryPick* we needed to perform several customizations (Section 3.5): i) selecting features of cloud configurations to minimize the search steps; ii) handling noise in the sampled data caused by cloud internal dynamics; iii) selecting initial samples; and iv) defining the stopping criteria.

We evaluate *CherryPick* on five popular analytical jobs with 66 configurations on Amazon EC2. *CherryPick* has a high chance (45%-90%) to pick the optimal configuration and otherwise can find a near-optimal solution (within 5% at the median), while alternative solutions such as coordinate descent and random search can take up to 75% more running time and 45% more search cost. We also compare *CherryPick* with Ernest [37] and show how *CherryPick* can improve search time by 90% and search cost by 75% for SQL queries.

2 Background and Motivation

In this section, we show the benefits and challenges of choosing the best cloud configurations. We also present two strawman solutions to solve this problem.

2.1 Benefits

A good cloud configuration can reduce the cost of analytic jobs by a large amount. Table 1 shows the arithmetic mean and maximum running cost of configurations compared to the configuration with minimum running cost for four applications across 66 candidate configurations. The details of these applications and their cloud configurations are described in Section 5. For example, for the big data benchmark, TPC-DS, the average configuration costs 3.4 times compared to the configuration with minimum cost; if users happen to choose the worst configuration, they would spend 9.6 times more.

Application	Avg/min	Max/min
TPC-DS	3.4	9.6
TPC-H	2.9	12
Regression (SparkML)	2.6	5.2
TeraSort	1.6	3.0

Table 1: Comparing the maximum, average, and minimum cost of configurations for various applications.

Picking a good cloud configuration is even more important for recurring jobs where similar workloads are executed repeatedly, e.g. daily log parsing. Recent studies report that up to 40% of analytics jobs are recurring [10, 17]. Our approach only works for repeating jobs, where the cost of a configuration search can be amortized across many subsequent runs.

2.2 Challenges

There are several challenges for picking the best cloud configurations for big data analytics jobs.

Complex performance model: The running time is affected by the amount of resources in the cloud configuration in a *non-linear* way. For instance, as shown in Figure 1, a regression job on SparkML (with fixed number of CPU cores) sees a diminishing return of running time at 256GB RAM. This is because the job does not benefit from more RAM beyond what it needs. Therefore, the running time only sees marginal improvements.

In addition, performance under a cloud configuration is not deterministic. In cloud environments, which is shared among many tenants, stragglers can happen. We measured the running time of TeraSort-30GB on 22 different cloud configurations on AWS EC2 five times. We then computed the coefficient of variation (CV) of the five runs. Our results show that the median of the CV is about 10% and the 90 percentile is above 20%. This variation is not new [17].

Cost model: The cloud charges users based on the amount of time the VMs are up. Using configurations with a lot of resources could minimize the running time, but it may cost a lot more money. Thus, to minimize cost, we have to find the right balance between resource prices and the running time. Figure 2 shows the cost of running Regression on SparkML on different cluster sizes where each VM comes with 15 GBs of RAM and 4 cores in AWS EC2. We can see that the cost does not monotonically increase or decrease when we add more resources into the cluster. This is because adding resources may accelerate the computation but also raises the price per unit of running time.

The heterogeneity of applications: Figure 3 shows different shapes for TPC-DS and Regression on Spark and how they relate to instance types. For TeraSort, a low memory instance (8 core and 15 GBs of RAM) performs the best because CPU is a more critical resource. On the other hand for Regression, the same cluster has 2.4 times more running time than the best candidate due to the lack of RAM.

Moreover, the best choice often depends on the application configurations, e.g., the number of map and reduce tasks in YARN. Our work on identifying the best cloud configurations is complementary to other works on identifying the best application configurations (e.g., [19, 38]). *CherryPick* can work with any (even not optimal) application configurations.

2.3 Strawman solutions

The two strawman solutions for predicting a near optimal cloud configuration are modeling and searching.

Accurate modeling of application performance. One way is to model application performance and then pick the best configuration based on this model. However, this methodology has poor adaptivity. Building a model that works for a variety of applications and cloud configurations can be difficult because the knowledge of the internal structure of specific applications is needed to make the model effective. Moreover, building a model through human intervention for every new application can be tedious.

Static searching for the best cloud configuration. Another way is to exhaustively search for the best cloud configuration without relying on an accurate performance model. However, this methodology has high overhead. With 40 instance types at Amazon EC2 and tens of cluster sizes for an application, if not careful, one could end up needing tens if not hundreds of runs to identify the best instance. In addition, trying each cloud configuration multiple times to get around the dynamics in the cloud (due to resource multiplexing and stragglers) would exacerbate the problem even further.

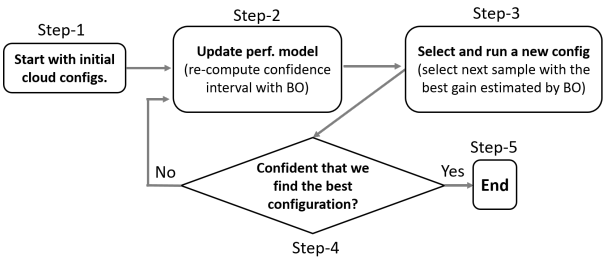


Figure 4: *CherryPick* workflow

To reduce the search time and cost, one could use *coordinate descent* and search one dimension at a time. Coordinate descent could start with searching for the optimal CPU/RAM ratio, then the CPU count per machine, then cluster size, and finally disk type. For each dimension, we could fix the other dimensions and search for the cheapest configuration possible. This could lead to suboptimal decisions if for example, because of bad application configuration a dimension is not fully explored or there are local minima in the problem space.

3 CherryPick Design

3.1 Overview

CherryPick follows a general principle in statistical learning theory [36]: "If you possess a restricted amount of information for solving some problem, try to solve the problem directly and never solve a more general problem as an intermediate step."

In our problem, the ultimate objective is to find the best configuration. We also have a very restricted amount of information, due to the limited runs of cloud configurations we can afford. Therefore, the model does not have enough information to be an accurate performance predictor, but this information is sufficient to find a good configuration within a few steps.

Rather than accurately predicting application performance, we just need a model that is accurate enough for us to separate the best configuration from the rest.

Compared to static searching solutions, we dynamically adapt our searching scheme based on the current understanding and confidence interval of the performance model. We can dynamically pick the next configuration that can best distinguish performance across configurations and eliminate unnecessary trials. The performance model can also help us understand when to stop searching earlier once we have a small enough confidence interval. Thus, we can reach the best configuration faster than static approaches.

Figure 4 shows the joint process of performance modeling and configuration searching. We start with a few initial cloud configurations (e.g., three), run them, and input the configuration details and job completion time into the performance model. We then *dynamically* pick the next cloud configuration to run based on the perfor-

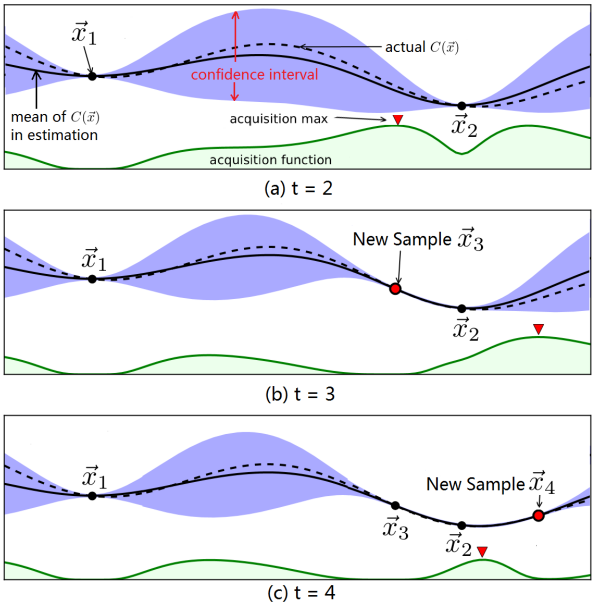


Figure 5: An example of BO's working process (derived from Figure 1 in [13]).

mance model and feed the result back to the performance model. We stop when we have enough confidence that we have found a good configuration.

3.2 Problem formulation

For a given application and workload, our goal is to find the optimal or a near-optimal cloud configuration that satisfies a performance requirement and minimizes the total execution cost. Formally, we use $T(\vec{x})$ to denote the running time function for an application and its input workloads. The running time depends on the cloud configuration vector $\vec{x}$, which includes instance family types, CPU, RAM, and other resource configurations.

Let $P(\vec{x})$ be the price per unit time for all VMs in cloud configuration $\vec{x}$. We formulate the problem as follows:

$$\begin{aligned} \underset{\vec{x}}{\text{minimize}} \quad & C(\vec{x}) = P(\vec{x}) \times T(\vec{x}) \\ \text{subject to} \quad & T(\vec{x}) \leq \mathscr{T}_{max} \end{aligned} \tag{1}$$

where $C(\vec{x})$ is the total cost of cloud configuration $\vec{x}$ and $\mathscr{T}_{max}$ is the maximum tolerated running time[1]. Knowing $T(\vec{x})$ under all candidate cloud configurations would make it straightforward to solve Eqn (1), but it is expensive because all candidate configurations need to be tried. Instead, we use BO (with Gaussian Process Priors, see Section 6) to directly search for an approximate solution of Eqn (1) with significantly smaller cost.

3.3 Solution with Bayesian Optimization

Bayesian Optimization (BO) [13, 28, 33] is a framework to solve optimization problem like Eqn. (1) where the objective function $C(\vec{x})$ is unknown beforehand but can be observed through experiments. By modeling $C(\vec{x})$ as a stochastic process, e.g. a Gaussian Process [26], BO can compute the *confidence interval* of $C(\vec{x})$ according to one or more samples taken from $C(\vec{x})$. A confidence interval is an area that the curve of $C(\vec{x})$ is most likely (e.g. with 95% probability) passing through. For example, in Figure 5(a), the dashed line is the actual function $C(\vec{x})$. With two samples at $\vec{x}_1$ and $\vec{x}_2$, BO computes a confidence interval that is marked with a blue shadowed area. The black solid line shows the expected value of $C(\vec{x})$ and the value of $C(\vec{x})$ at each input point $\vec{x}$ falls in the confidence interval with 95% probability. The confidence interval is updated (posterior distribution in Bayesian Theorem) after new samples are taken at $\vec{x}_3$ (Figure 5(b)) and $\vec{x}_4$ (Figure 5(c)), and the estimate of $C(\vec{x})$ improves as the area of the confidence interval decreases.

BO can smartly decide the next point to sample using a pre-defined *acquisition function* that also gets updated with the confidence interval. As shown in Figure 5, $\vec{x}_3$ ($\vec{x}_4$) is chosen because the acquisition function at $t = 2$ ($t = 3$) indicates that it has the most potential gain. There are many designs of acquisition functions in the literature, and we will discuss how we chose among them in Section 3.5.

BO is embedded into *CherryPick* as shown in Figure 4. At Step 2, *CherryPick* leverages BO to update the confidence interval of $C(\vec{x})$. After that, at Step 3, *CherryPick* relies on BO's acquisition function to choose the best configuration to run next. Also, at Step 4, *CherryPick* decides whether to stop the search according to the confidence interval of $C(\vec{x})$ provided by BO (details shown in Section 3.5).

Another useful property of BO is that it can accommodate observation noise in the computation of confidence interval of the objective function. Suppose in practice, given an input point $\vec{x}$, we have no direct access to $C(\vec{x})$ but can only observe $C(\vec{x})'$ that is:

$$C(\vec{x})' = C(\vec{x}) + \varepsilon \tag{2}$$

where ε is a Gaussian noise with zero mean, that is $\varepsilon \sim \mathscr{N}(0, \sigma_\varepsilon^2)$. Because $C(\vec{x})'$ is also Gaussian, BO is able to infer the confidence interval of $C(\vec{x})$ according to the samples of $C(\vec{x})'$ and ε [13]. Note that in our scenario, the observation noise on $C(\vec{x})$ is negligible because the measurement on running time and price model is accurate enough. However, the ability to handle the additive noise of BO is essential for us to handle the uncertainty in clouds (details in Section 3.6).

In summary, by integrating BO, *CherryPick* has the ability to learn the objective function quickly and only take samples in the areas that most likely contain the minimum point. For example, in Figure 5(c) both $\vec{x}_3$ and $\vec{x}_4$ are close to the minimum point of the actual $C(\vec{x})$,

[1] $C(\vec{x})$ assumes a fixed number of identical VMs.

leaving the interval between $\vec{x_1}$ and $\vec{x_4}$ unexplored without any impact on the final result.

3.4 Why do we use Bayesian Optimization?

BO is effective in finding optimal cloud configurations for Big Data analytics for three reasons.

First, BO does not limit the function to be of any predefined format, as it is non-parametric. This property makes *CherryPick* useful for a variety of applications and cloud configurations.

Second, BO typically needs a small number of samples to find a near-optimal solution because BO focuses its search on areas that have the largest expected improvements.

Third, BO can tolerate uncertainty. *CherryPick* faces two main sources of uncertainty: (i) because of the small number of samples, *CherryPick*'s performance models are imperfect and usually have substantial prediction errors; (ii) the cloud may not report a stable running time even for the same application due to resource multiplexing across applications, stragglers, etc. BO can quantitatively define the uncertainty region of the performance model. The confidence interval it computes can be used to guide the searching decisions even in face of model inaccuracy. In Section 3.6, we leverage this property of BO to handle the uncertainty from cloud dynamics.

One limitation of BO is that its computation complexity is $O(N^4)$, where N is the number of data samples. However, this is perfectly fine because our data set is small (our target is typically less than 10 to 20 samples out of hundreds of candidate cloud configurations).

Alternatives Alternative solutions often miss one of the above benefits: (1) linear regression and linear reinforcement learning are not generic to all applications because they do not work for non-linear models; (2) techniques that try to model a function (e.g., linear regression, support vector regression, boosting tree, etc.) do not consider minimizing the number of sample points. Deep neural networks [27], table-based modeling [11], and Covariance matrix adaptation evolution strategy (CMA-ES) [25] can potentially be used for black-box optimization but require a large number of samples. (3) It is difficult to adapt reinforcement learning [27, 35] to handle uncertainty and minimize the number of samples while BO models the uncertainty so as to accelerate the search.

3.5 Design options and decisions

To leverage Bayesian Optimization to find a good cloud configuration, we need to make several design decisions based on system constraint and requirements.

Prior function As most BO frameworks do, we choose to use Gaussian Process as the prior function. It means that we assume the final model function is a sample from Gaussian Process. We will discussion this choice in more details in Section 6.

We describe $C(\vec{x})$ with a mean function $\mu(\cdot)$ and covariance kernel function $k(\cdot,\cdot)$. For any pairs of input points $\vec{x}_1, \vec{x}_2$, we have:

$$\mu(\vec{x}_1) = \mathbb{E}[C(\vec{x}_1)]; \mu(\vec{x}_2) = \mathbb{E}[C(\vec{x}_2)]$$
$$k(\vec{x}_1,\vec{x}_2) = \mathbb{E}[(C(\vec{x}_1) - \mu(\vec{x}_1))(C(\vec{x}_2) - \mu(\vec{x}_2))]$$

Intuitively, we know that if two cloud configurations, $\vec{x}_1$ and $\vec{x}_2$ are similar to each other, $C(\vec{x}_1)$ and $C(\vec{x}_2)$ should have large covariance, and otherwise, they should have small covariance. To express this intuition, people have designed numerous formats of the covariance functions between inputs $\vec{x}_1$ and $\vec{x}_2$ which decrease when $||\vec{x}_1 - \vec{x}_2||$ grow. We choose *Matern5/2* [31] because it does not require strong smoothness and is preferred to model practical functions [33].

Acquisition function There are three main strategies to design an acquisition function [33]: (i) Probability of Improvement (PI) – picking the point which can maximize the probability of improving the current best; (ii) Expected Improvement (EI) – picking the point which can maximize the expected improvement over the current best; and (iii) Gaussian Process Upper Confidence Bound (GP-UCB) – picking the point whose certainty region has the smallest lower bound(when we minimize a function). In *CherryPick* we choose EI [13] as it has been shown to be better-behaved than PI, and unlike the method of GP-UCB, it does not require its own tuning parameter [33].

Jones et al. [22] derive an easy-to-compute closed form for the EI acquisition function. Let X_t be the collection of all cloud configurations whose function values have been observed by round t, and $m = \min_{\vec{x}}\{C(\vec{x})|\vec{x} \in X_t\}$ as the minimum function value observed so far. For each input $\vec{x}$ which is not observed yet, we can evaluate its expected improvement if it is picked as the next point to observe with the following equation:

$$EI(\vec{x}) = \begin{cases} (m - \mu(\vec{x}))\Phi(Z) + \sigma(\vec{x})\phi(Z), & \text{if } \sigma(\vec{x}) > 0 \\ 0, & \text{if } \sigma(\vec{x}) = 0 \end{cases} \quad (3)$$

where $\sigma(\vec{x}) = \sqrt{k(\vec{x},\vec{x})}$, $Z = \frac{m-\mu(\vec{x})}{\sigma(\vec{x})}$, and Φ and ϕ are standard normal cumulative distribution function and the standard normal probability density function respectively.

The acquisition function shown in Eqn (3) is designed to minimize $C(\vec{x})$ without further constraints. Nonetheless, from Eqn 1 we know that we still have a performance constraint $T(\vec{x}) \leq \mathcal{T}_{max}$ to consider. It means that when we choose the next cloud configuration to evaluate, we should have a bias towards one that is likely to satisfy the performance constraint. To achieve this goal, we first build the model of running time function $T(\vec{x})$ from $\frac{C(\vec{x})}{P(\vec{x})}$.

Then, as suggested in [18], we modify the EI acquisition function as:

$$EI(\vec{x})' = P[T(\vec{x}) \leq \mathscr{T}_{max}] \times EI(\vec{x}) \quad (4)$$

Stopping condition We define the stopping condition in *CherryPick* as follows: when the expected improvement in Eqn.(4) is less than a threshold (e.g. 10%) and at least N (e.g. $N = 6$) cloud configurations have been observed. This ensures that *CherryPick* does not stop the search too soon and it prevents *CherryPick* from struggling to make small improvements.

Starting points Our choice of starting points should give BO an estimate about the shape of the cost model. For that, we sample a few points (e.g., three) from the sample space using a quasi-random sequence [34]. Quasi-random numbers cover the sample space more uniformly and help the prior function avoid making wrong assumptions about the sample space.

Encoding cloud configurations We encode the following features into $\vec{x}$ to represent a cloud configuration: the number of VMs, the number of cores, CPU speed per core, average RAM per core, disk count, disk speed and network capacity of a VM.

To reduce the search space of the Bayesian Optimization, we normalize and discretized most of the features. For instance, for disk speed, we only define fast and slow to distinguish SSD and magnetic disks. Similarly, for CPU, we use fast and slow to distinguish high-end and common CPUs. Such discretization significantly reduces the space of several features without losing the key information brought by the features and it also helps to reduce the number of invalid cloud configurations. For example, we can discretize the space so that the CPUs greater (smaller) than 2.2GHz are fast (slow) and the disks with bandwidth greater (smaller) than 600MB/s are fast (slow). Then, if we suggest a (fast, fast) combination for (CPU, Disk), we could choose a 2.5Ghz and 700MBs instance (or any other one satisfying the boundary requirements). Or in place of a (slow, slow) configuration we could pick an instance with 2Ghz of speed and 400MB/s of IO bandwidth. If no such configurations exist, we can either remove that point from the candidate space that BO searches or return a large value, so that BO avoids searching in that space.

3.6 Handling uncertainties in clouds

So far we assumed that the relation between cloud configurations and cost (or running time) is deterministic. However, in practice, this assumption can be broken due to uncertainties within any shared environment. The resources of clouds are shared by multiple users so that different users' workload could possibly have interference with each other. Moreover, failures and resource overloading, although potentially rare, can impact the completion time of a job. Therefore, even if we run the same workload on the same cloud with the same configuration for multiple times, the running time and cost we get may not be the same.

Due to such uncertainties in clouds, the running time we can observe from an actual run on configuration $\vec{x}$ is $\tilde{T}(\vec{x})$ and the cost is $\tilde{C}(\vec{x})$. If we let $T(\vec{x}) = \mathbb{E}[\tilde{T}(\vec{x})]$ and $C(\vec{x}) = \mathbb{E}[\tilde{C}(\vec{x})]$, we have:

$$\tilde{T}(\vec{x}) = T(\vec{x})(1+\varepsilon_c) \quad (5)$$

$$\tilde{C}(\vec{x}) = C(\vec{x})(1+\varepsilon_c) \quad (6)$$

where ε_c is a multiplicative noise introduced by the uncertainties in clouds. We model ε_c as normally distributed: $\varepsilon_c \sim \mathcal{N}(0, \sigma^2_{\varepsilon_c})$.

Therefore, Eqn (1) becomes minimizing the expected cost with the expected performance satisfying the constraint.

BO cannot infer the confidence interval of $C(\vec{x})$ from the observation of $\tilde{C}(\vec{x})$ because the latter is not normally distributed given that BO assumes $C(\vec{x})$ is Gaussian and so is $(1+\varepsilon_c)$. One straightforward way to solve this problem is to take multiple samples at the same configuration $\vec{x}$, so that $C(\vec{x})$ can be obtained from the average of the multiple $\tilde{C}(\vec{x})$. Evidently, this method will result in a big overhead in search cost.

Our key idea to solve this problem (so that we only take one sample at each input) is to transform Eqn. (1) to the following equivalent format:

$$\begin{aligned} \underset{\vec{x}}{\text{minimize}} \quad & \log C(\vec{x}) = \log P(\vec{x}) + \log T(\vec{x}) \\ \text{subject to} \quad & \log T(\vec{x}) \leq \log \mathscr{T}_{max} \end{aligned} \quad (7)$$

We use BO to minimize $\log C(\vec{x})$ instead of $C(\vec{x})$ since:

$$\log \tilde{C}(\vec{x}) = \log C(\vec{x}) + \log(1+\varepsilon_c) \quad (8)$$

Assuming that ε_c is less than one (e.g. $\varepsilon_c < 1$), $\log(1+\varepsilon_c)$ can be estimated by ε_c, so that $\log(1+\varepsilon_c)$ can be viewed as an observation noise with a normal distribution, and $\log \tilde{C}(\vec{x})$ can be treated as the observed value of $\log C(\vec{x})$ with observation noise. Eqn.(8) can be solved similar to Eqn.(2).

In the implementation of *CherryPick*, we use Eqn. (7) instead of Eqn. (1) as the problem formulation.

4 Implementation

In this section, we discuss the implementation details of *CherryPick* as shown in Figure 6. It has four modules.

1. Search Controller: Search Controller orchestrates the entire cloud configuration selection process. To use *CherryPick*, users supply a representative workload (see Section 6) of the application, the objective (e.g. minimizing cost or running time), and the constraints (e.g. cost budget, maximum running time, preferred instance

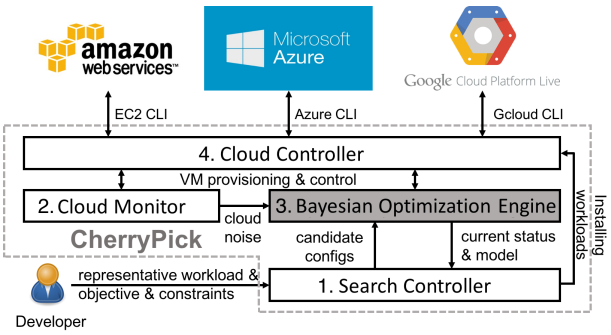


Figure 6: Architecture of *CherryPick*'s implementation.

types, maximum/minimum cluster size, etc.). Based on these inputs, the search controller obtains a list of candidate cloud configurations and passes it to the Bayesian Optimization Engine. At the same time, Search Controller installs the representative workload to clouds via Cloud Controller. This process includes creating VMs in each cloud, installing the workload (applications and input data), and capturing a customized VM image which contains the workload. Search Controller also monitors the current status and model on the Bayesian Optimization engine and decides whether to finish the search according to the stopping condition discussed in Section 3.5.

2. Cloud Monitor: Cloud Monitor runs benchmarking workloads of Big Data defined by *CherryPick* on different clouds. It repeats running numerous categories of benchmark workloads on each cloud to measure the upper-bound (or high percentile) of the cloud noise [2]. The result is offered to Bayesian Optimization engine as the ε_c in Eqn. (8). This monitoring is lightweight; we only need to run this system every few hours with a handful of instances.

3. Bayesian Optimization Engine: Bayesian Optimization Engine is built on top of Spearmint [6] which is an implementation of BO in Python. Besides the standard BO, it also has realized our acquisition function in Eqn (3) and the performance constraint in Eqn (4). However, Spearmint's implementation of Eqn (4) is not efficient for our scenario because it assumes $C(\vec{x})$ and $T(\vec{x})$ are independent and trains the models of them separately. We modified this part so that $T(\vec{x})$ is directly derived from $\frac{C(\vec{x})}{P(\vec{x})}$ after we get the model of $C(\vec{x})$. Our implementation of this module focuses on the interfaces and communications between this module and others. For taking a sample of a selected cloud configuration, the BO engine submits a cluster creation request and a start workload request via the Cloud Controller.

4. Cloud Controller: Cloud Controller is an adaptation layer which handles the heterogeneity to control the clouds. Each cloud has its own APIs and semantics to do the operations such as create/delete VMs, create/delete virtual networks, capturing images from VMs, and list the available instance types. Cloud Controller defines a uniform API for the other modules in *CherryPick* to perform these operations. In addition, the API also includes sending commands directly to VMs in clouds via SSH, which facilitates the control of the running workload in the clouds.

The entire *CherryPick* system is written in Python with about 5,000 lines of code, excluding the legacy part of Spearmint.

[2] Over-estimating ε_c means more search cost.

5 Evaluation

We evaluate *CherryPick* with 5 types of big data analytics applications on 66 cloud configurations. Our evaluations show that *CherryPick* can pick the optimal configuration with a high chance (45-90%) or find a near-optimal configuration (within 5% of the optimal at the median) with low search cost and time, while alternative solutions such as coordinate descent and random search can reach up to 75% more running time and up to 45% more search time than *CherryPick*. We also compare *CherryPick* with Ernest [37] and show how *CherryPick* can reduce the search time by 90% and search cost by 75% for SQL queries. We discuss insights on why *CherryPick* works well and show how *CherryPick* adapt to changing workloads and various performance constraints.

5.1 Experiment setup

Applications: We chose benchmark applications on Spark [44] and Hadoop [41] to exercise different CPU/Disk/RAM/Network resources: *(1) TPC-DS* [7] is a recent benchmark for big data systems that models a decision support workload. We run TPC-DS benchmark on Spark SQL with a scale factor of 20. *(2) TPC-H* [8] is another SQL benchmark that contains a number of ad-hoc decision support queries that process large amounts of data. We run TPC-H on Hadoop with a scale factor of 100. Note that our trace runs 20 queries concurrently. While it may be possible to model each query's performance, it is hard to model the interactions of these queries together. *(3) TeraSort* [29] is a common benchmarking application for big data analytics frameworks [1, 30], and requires a balance between high IO bandwidth and CPU speed. We run TeraSort on Hadoop with 300 GB of data, which is large enough to exercise disks and CPUs together. *(4) The SparkReg* [4] benchmark consists of machine learning workloads implemented on top of Spark. We ran the regression workload in SparkML with 250k examples, 10k features, and 5 iterations. This workload heavily depends on memory space for caching data and has minimal use for disk IO. *(5) SparkKm* is another SparkML benchmark [5]. It is

Instance Size	Number of instances					
large	16	24	32	40	48	56
xlarge	8	12	16	20	24	28
2xlarge	4	6	8	10	12	14
Number of Cores	32	48	64	80	96	112

Table 2: Configurations for one instance family.

a clustering algorithm that partitions a space into k clusters with each observation assigned to the cluster with the closest mean. We use 250k observations with 10k features. Similar to SparkReg, this workload is dependent on memory space and has less stringent requirements for CPU and disk IO.

Cloud configurations: We choose four families in Amazon EC2: M4 (general purpose), C4 (compute optimized), R3 (memory optimized), I2 (disk optimized) instances. Within each family, we used large, xlarge, and 2xlarge instance sizes each with 2, 4, and 8 cores per machine respectively. For each instance size, we change the total number of cores from 32 to 112. Table 2 shows the 18 configurations for each of the four families. We run a total of 66 configurations, rather than 72 (18×4), because the smallest size for I2 starts from xlarge. We do not choose more configurations due to time and expense constraints. However, we make sure the configurations we choose are reasonable for our objective (i.e., minimizing cost). For example, we did not choose 4xlarge instances because 4xlarge is more expensive than 2xlarge but shows diminishing returns in terms of running time.

Objectives: We define the objective as minimizing the cost of executing the application under running time constraints. By default, we set a loose constraint for running time so *CherryPick* searches through a wider set of configurations. We evaluate tighter constraints in Section 5.4. Note that minimizing running time with no cost constraint always leads to larger clusters, and therefore, is rather simple. On the other hand, minimizing the cost depends on the right balance between cluster size and cluster utilization.

***CherryPick* settings:** By default, we use EI= 10%, $N = 6$, and 3 initial samples. In our experiments, we found that EI= 10% gives a good trade-off between search cost and accuracy. We also tested other EI values in one experiment.

Alternative solutions: We compare *CherryPick* with the following strategies: *(1) Exhaustive search*, which finds the best configuration by running all the configurations; *(2) Coordinate descent*, which searches one coordinate – in order of CPU/RAM ratio (which specifies the instance family type), CPU count, cluster size, disk type – at a time (Section 2.3) from a randomly chosen starting point. The ordering of dimensions can also impact the result. It is unclear whether a combination of dimensions and ordering exists that works best across all applications. Similar approaches have been used for tuning configurations for Map Reduce jobs and web servers [24, 42]. *(3) Random search with a budget*, which randomly picks a number of configurations given a search budget. Random search is used by previous configuration tuning works [20, 43]. *(4) Ernest [37]*, which builds a performance model with common communication patterns. We run Ernest once per-instance type and use the model to predict the optimal number of instances.

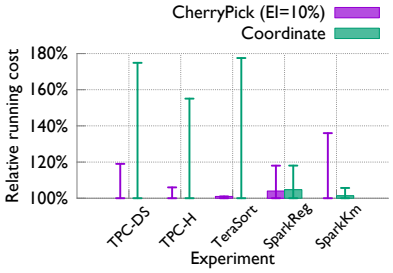
Metrics: We compare *CherryPick* with alternative solutions using two metrics: (i) the running cost of the configuration: the expense to run a job with the selected configuration; (ii) the search cost: the expense to run all the sampled configurations. All the reported numbers are normalized by the exhaustive search cost and running cost across the clusters in Table 2.

We run *CherryPick* and random search 20 times with different seeds for starting points. For the coordinate descent, we start from all the 66 possible starting configurations. We then show the 10th, median, and 90th percentile of the search cost and running cost of *CherryPick* normalized by the optimal configuration reported by exhaustive search.

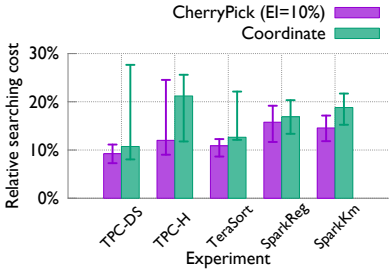
5.2 Effectiveness of *CherryPick*

***CherryPick* finds the optimal configuration in a high chance (45-90%) or a near-optimal configuration with low search cost and time:** Figure 7a shows the median, 10th percentile, and 90th percentile of running time for the configuration picked by *CherryPick* for each of the five workloads. *CherryPick* finds the exact optimal configuration with 45-90% chance, and finds a configuration within 5% of the optimal configuration at the median. However, using exhaustive search requires 6-9 times more search cost and 5-9.5 times more search time compared with *CherryPick*. On AWS, which charges on an hourly basis, after running TeraSort 100 times, exhaustive search costs $581 with $49 for the remainder of the runs. While *CherryPick* uses $73 for searching and $122 for the rest of the runs saving a total of $435.

In terms of accuracy, we find that that *CherryPick* has good accuracy across applications. On median, *CherryPick* finds an optimal configuration within 5% of the optimal configuration. For TPC-DS, *CherryPick* finds a configuration within 20% of the optimal in the 90th percentile; For TPC-H, the 90th percentile is 7% worse than optimal configuration; Finally, for TeraSort, SparkReg, and SparkKm *CherryPick*'s 90th percentile configuration is 0%, 18%, 38% worse than the optimal respectively. It is possible to change the EI of *CherryPick* to find even better configurations.



(a) Running cost



(b) Search cost

Figure 7: Comparing *CherryPick* with coordinate descent. The bars show 10th and 90th percentile.

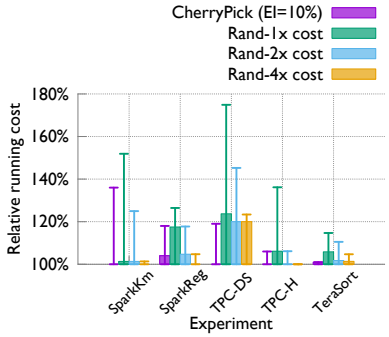


Figure 8: Running cost of configurations by *CherryPick* and random search. The bars show 10th and 90th percentile.

***CherryPick* is more stable in picking near-optimal configurations and has less search cost than coordinate descent.** Across applications, the median configuration suggested by coordinate descent is within 7% of the optimal configuration. On the other hand, the tail of the configuration suggested by coordinate descent can be far from optimal. For TPC-DS, TPC-H, and TeraSort, the tail configuration is 76%, 56%, and 78% worse than optimal, while using comparable or more search cost. This is because coordinate descent can be misled by the result of the run. For example, for TPC-DS, C4 family type has the best performance. In our experiment, if coordinate descent starts its search from a configuration with a large number of machines, the C4 family fails to finish the job successfully due to the scheduler failing. Therefore, the C4 family is never considered in the later iterations of coordinate descent runs. This leads coordinate descent to a suboptimal point that can be much worse than the optimal configuration.

In contrast, *CherryPick* has stronger ability to navigate around these problems because even when a run fails to finish on a candidate configuration, it uses Gaussian process to model the global behavior of the function from the sampled configurations.

***CherryPick* reaches better configurations with more stability compared with random search with similar budget:** Figure 8 compares the running cost of configurations suggested by *CherryPick* and random search with equal/2x/4x search cost. With the same search cost, random search performs up to 25% worse compared to *CherryPick* on the median and 45% on the tail. With 4x cost, random search can find similar configurations to *CherryPick* on the median. Although *CherryPick* may end up with different configurations with different starting points, it consistently has a much higher stability of the running cost compared to random search. *CherryPick* has a comparable stability to random search with 4x budget, since random search with a 4x budget almost visits all the configurations at least once.

***CherryPick* reaches configurations with similar running cost compared with Ernest [37], but with lower search cost and time:** It is hard to extend Ernest to work with a variety of applications because it requires using a small representative dataset to build the model. For example, TPC-DS contains 99 queries on 24 tables, where each query touches a different set of tables. This makes it difficult to determine which set of tables should be sampled to build a representative small-scale experiment. To overcome this we use the TPC-DS data generator and generate a dataset with scale factor 2 (10% of target data size) and use that for training. We then use Ernest to predict the best configuration for the target data size. Finally, we note that since Ernest builds a separate model for each instance type we repeat the above process 11 times, once for each instance type.

Figure 9 shows that Ernest picks the best configuration for TPC-DS, the same as *CherryPick*, but takes 11 times the search time and 3.8 times the search cost. Although Ernest identifies the best configuration, its predicted running time is up to 5 times of the actual running time. This is because, unlike iterative ML workloads, the TPC-DS performance model has a complex scaling behavior with input scale and this is not captured by the linear model used in Ernest. Thus, once we set a tighter performance constraint, Ernest suggests a configuration that is 2 times more expensive than *CherryPick* with 2.8 times more search cost.

***CherryPick* can tune EI to trade-off between search cost and accuracy:** The error of the tail configuration for SparkKm as shown in Figure 7a can be as high as 38%. To get around this problem, the users can use lower values of EI to find better configurations. Figure 10 shows the running cost and search cost for different values of EI. At $EI < 6\%$, *CherryPick* has much

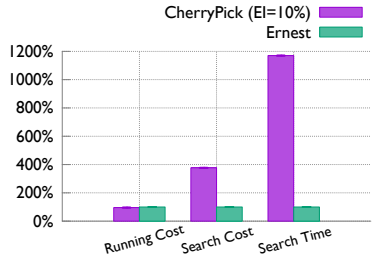


Figure 9: Comparing Ernest to *CherryPick* (TPC-DS).

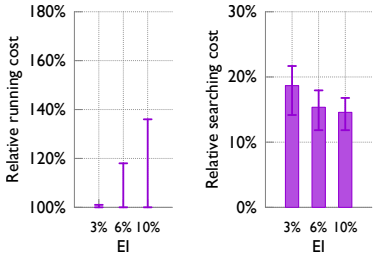


Figure 10: Search cost and running cost of SparkKm with different EI values.

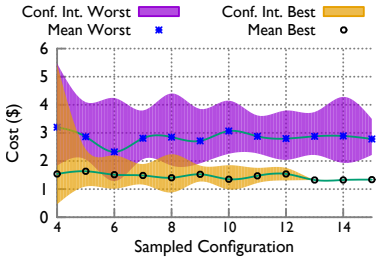
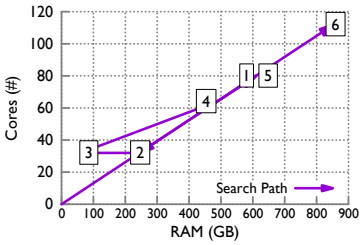
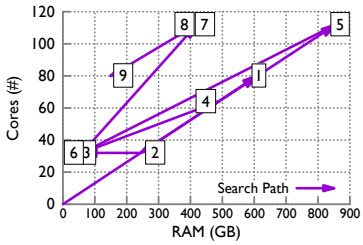


Figure 11: Bayesian opt. process for the best/worst configuration (TeraSort).



(a) SparkReg



(b) TPC-DS

	SparkReg		TPC-DS	
Step	**VM Type**	**# VMs**	**VM Type**	**# VMs**
1	r3.2xlarge	10	r3.2xlarge	10
2	r3.2xlarge	4	r3.2xlarge	4
3	c4.xlarge	8	c4.xlarge	8
4	r3.large	32	r3.large	32
5	i2.2xlarge	10	r3.2xlarge	14
6	r3.2xlarge	14	c4.2xlarge	4
7			m4.xlarge	28
8			m4.2xlarge	14
9			c4.2xlarge	10

(c) Search path for TPC-DS and SparkReg

Figure 12: Search path for TPC-DS and SparkReg

better accuracy, finding configurations that at 90th percentile are within 18% of the optimal configuration. If we set $EI < 3\%$, *CherryPick* suggests configurations that are within 1% of the optimal configuration at 90th percentile resulting in a 26% increase in search cost.

This can be a knob where users of *CherryPick* can trade-off optimality for search cost. For example, if users of *CherryPick* predict that the recurring job will be popular, setting a low EI value can force *CherryPick* to look for better configurations more carefully. This may result in larger savings over the lifetime of the job.

5.3 Why *CherryPick* works?

We now show *CherryPick* behavior matches the key insights discussed in Section 3.1. For this subsection, we set the stopping condition EI<1% to make it easier to show how *CherryPick* navigates the space.

Previous performance prediction solutions require many training samples to improve prediction accuracy. *CherryPick* spends the budget to improve the prediction accuracy of those configurations that are closer to the best. Figure 11 shows the means and confidence intervals of the running cost for the best and worst configurations, and how the numbers change during the process of Bayesian optimization. Initially, both configurations have large confidence intervals. As the search progresses, the confidence interval for the best configuration narrows. In contrast, the estimated cost for the worst configuration has a larger confidence interval and remains large. This is because *CherryPick* focuses on improving the estimation for configurations that are closer to the optimal.

Although *CherryPick* takes a black-box approach, it automatically learns the relation between cloud resources and the running time. Figure 13 shows *CherryPick*'s final estimation of the running time versus cluster size. The real curve follows Amdahl's law: (1) adding more VMs reduces the running time; (2) at some point, adding more machines has diminishing returns due to the sequential portion of the application. The real running time falls within the confidence interval of *CherryPick*. Moreover, *CherryPick* has smaller confidence intervals for the more promising region where the best configurations (those with more VMs) are located. It does not bother to improve the estimation for configurations with fewer VMs.

Even though *CherryPick* has minimal information about the application, it adapts the search towards the features that are more important to the application. Figure 12 shows example search paths for TPC-DS and SparkReg from the same three starting configurations. For SparkReg, *CherryPick* quickly identifies that clusters with larger RAM (R3 instances) have better performance and redirects the search towards such instances. In contrast, for TPC-DS, the last few steps suggest that *CherryPick* has identified that CPU is more important, and therefore the exploration is directed towards VMs with better CPUs (C4 instances). Figure 12 shows that *CherryPick* directly searches more configurations with larger #cores for TPC-DS than for SparkReg.

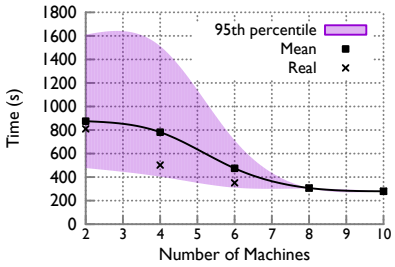


Figure 13: *CherryPick* learns diminishing returns of larger clusters (TPC-H, c4.2xlarge VMs).

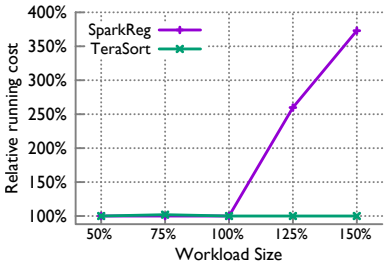


Figure 14: Sensitivity to workload size

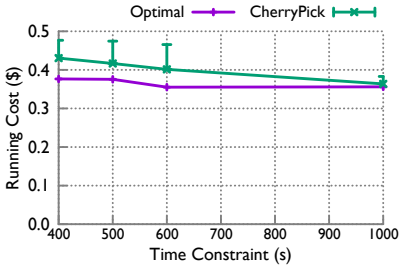


Figure 15: *CherryPick* works with time constraints (TPC-H).

5.4 Handling workload changes

CherryPick depends on representative workloads. Thus, one concern is *CherryPick*'s sensitivity to the variation of input workloads. In Figure 14, we keep the best configuration for the original workload (100% input size) C_{100} and test the running cost of the $C_{100\%}$ on workloads with 50% to 150% of the original input size. For TeraSort, we can continue to use $C_{100\%}$ to achieve the optimal cost with different input sizes. For SparkReg, $C_{100\%}$ remains effective for smaller workloads. However, when the workload is increased by 25%, $C_{100\%}$ can get to 260% the running cost of the new best configuration ($C_{125\%}$). This is because $C_{100\%}$ does not have enough RAM for SparkReg, which leads to more disk accesses.

Since input workloads usually vary in practice, *CherryPick* needs a good selection of representative workloads For example, for SparkReg, we should choose a relatively larger workload as the representative workload (e.g., choosing 125% gives you more stability than choosing 100%). We will discuss more on how to select representative workloads in Section 6.

When the difference between *CherryPick*'s estimation of the running cost and the actual running cost is above a threshold, the user can rerun *CherryPick*. For example, in Figure 14, suppose the user trains *CherryPick* with a 100% workload for SparkReg. With a new workload at size 125%, when he sees the running cost becomes 2x higher than expected, he can rerun *CherryPick* to build a new model for the 125% workload.

5.5 Handling performance constraints

We also evaluate *CherryPick* with tighter performance constraints on the running time (400 seconds to 1000 seconds) for TPC-H, as shown in Figure 15. *CherryPick* consistently identifies near-optimal configuration (2-14% difference with the optimal) with similar search cost to the version without constraints.

6 Discussion

Representative workloads: *CherryPick* relies on representative workloads to learn and suggest a good cloud configuration for similar workloads. Two workloads are similar if they operate on data with similar structures and sizes, and the computations on the data are similar. For example, for recurring jobs like parsing daily logs or summarizing daily user data with the same SQL queries, we can select one day's workload to represent the following week or month, if in this period the user data and the queries are not changing dramatically. Many previous works were built on top of the similarity in recurring jobs [10, 17]. Picking a representative workload for non-recurring jobs hard, and for now, *CherryPick* relies on human intuitions. An automatic way to select representative workload is an interesting avenue for future work.

The workload for recurring jobs can also change with time over a longer term. *CherryPick* detects the need to recompute the cloud configuration when it finds large gaps between estimated performance and real performance under the current configuration.

Larger search space: With the customizable virtual machines [2] and containers, the number of configurations that users can run their applications on becomes even larger. In theory, the large candidate number should not impact on the complexity of *CherryPick* because the computation time is only related with the number of samples rather than the number of candidates (BO works even in continuous input space). However, in practice, it might impact the speed of computing the maximum point of the acquisition function in BO because we cannot simply enumerate all of the candidates then. More efficient methods, e.g. Monte Carlo simulations as used in [6], are needed to find the maximum point of the acquisition function in an input-agnostic way. Moreover, the computations of acquisition functions can be parallelized. Hence, customized VM only has small impacts on the feasibility and scalability of *CherryPick*.

Choice of prior model: By choosing Gaussian Process as a prior, we assume that the final function is a sample from Gaussian Process. Since Gaussian Process is non-parametric, it is flexible enough to approach the actual function given enough data samples. The closer the ac-

tual function is to a Gaussian Process, the fewer the data samples and searching we need. We admit that a better prior might be found given some domain knowledge of specific applications, but it also means losing the automatic adaptivity to a set of broader applications.

Although any *conjugate distribution* can be used as a prior in BO [32], we chose Gaussian Process because it is widely accepted as a good surrogate model for BO [33]. In addition, when the problem scale becomes large, Gaussian Process is the only choice which is computationally tractable as known so far.

7 Related Work

Current practices in selecting cloud configurations Today, developers have to select cloud configurations based on their own expertise and tuning. Cloud providers only make high-level suggestions such as recommending I2 instances in EC2 for IO intensive applications, e.g., Hadoop MapReduce. However, these suggestions are not always accurate for all workloads. For example, for our TPC-H and TeraSort applications on Hadoop MapReduce, I2 is not always the best instance family to choose.

Google provides recommendation services [3] based on the monitoring of average resource usage. It is useful for saving cost but is not clear how to adjust the resource allocation (e.g. scaling down VMs vs. reducing the cluster size) to guarantee the application performance.

Selecting cloud configurations for specific applications The closest work to us is Ernest [37], which we have already compared in Section 1. We also have discussed previous works and strawman solutions in Section 2 that mostly focus on predicting application performance [19, 21, 37]. Bodik *et al*. [12] proposed a framework that learns performance models of web applications with lightweight data collection from a production environment. It is not clear how to use such data collection technique for modeling big data analytics jobs, but it is an interesting direction we want to explore in the future.

Previous works [11, 40] leverage table based models to predict performance of applications on storage devices. The key idea is to build tables based on input parameters and use interpolation between tables for prediction. However, building such tables requires a large amount of data. While such data is available to data center operators, it is out of reach for normal users. *CherryPick* works with a restricted amount of data to get around this problem.

Tuning application configurations: There are several recent projects that have looked at tuning application configurations within fixed cloud environments. Some of them [19, 38, 45] propose to monitor resource usage in Hadoop framework and adjust Hadoop configurations to improve the application performance. Others search for the best configurations using random search [19] or local search [24, 42]. Compared to Hadoop configuration, cloud configurations have a smaller search space but a higher cost of trying out a configuration (both the expense and the time to start a new cluster). Thus we find Bayesian optimization a better fit for our problem. *CherryPick* is complementary to these works and can work with any application configurations.

Online scheduler of applications: Paragon [15] and Quasar [16] are online schedulers that leverage historical performance data from scheduled applications to quickly classify any new incoming application, assign the application proper resources in a datacenter, and reduce interferences among different applications. They also rely on online adjustments of resource allocations to correct mistakes in the modeling phase. The methodology cannot be directly used in *CherryPick*'s scenarios because usually, users do not have historical data, and online adjustment (e.g., changing VM types and cluster sizes) is slow and disruptive to big data analytics. Containers allow online adjustment of system resources, so it might be worth revisiting these approaches.

Parameter tuning with BO: Bayesian Optimization is also used in searching optimal Deep Neural Network configurations for specific Deep Learning workloads [9, 33] and tuning system parameters [14]. *CherryPick* is a parallel work which searches cloud configurations for big data analytics.

8 Conclusion

We present *CherryPick*, a service that selects near-optimal cloud configurations with high accuracy and low overhead. *CherryPick* adaptively and automatically builds performance models for specific applications and cloud configurations that are *just accurate enough* to distinguish the optimal or a near-optimal configuration from the rest. Our experiments on Amazon EC2 with 5 widely used benchmark workloads show that *CherryPick* selects optimal or near-optimal configurations with much lower search cost than existing solutions.

9 Acknowledgments

We like to thank our shepherd, John Wilkes, for his extensive comments. John's thoughtful interaction has substantially improved the presentation of this work. Further, thanks to Jiaqi Gao, Yuliang Li, Mingyang Zhang, Luis Pedrosa, Behnaz Arzani, Wyatt Lloyd, Victor Bahl, Srikanth Kandula, and the NSDI reviewers for their comments on earlier drafts of this paper. This research is partially supported by CNS-1618138, CNS-1453662, CNS-1423505, and Facebook.

References

[1] Apache Spark the fastest open source engine for sorting a petabyte. https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html.

[2] Custom Machine Types - Google Cloud Platform. https://cloud.google.com/custom-machine-types/.

[3] Google VM rightsizing service. https://cloud.google.com/compute/docs/instances/viewing-sizing-recommendations-for-\instances.

[4] Performance tests for Spark. https://github.com/databricks/spark-perf.

[5] Performance tests for spark. https://github.com/databricks/spark-perf.

[6] Spearmint. https://github.com/HIPS/Spearmint.

[7] TPC Benchmark DS. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.3.0.pdf.

[8] TPC Benchmark H. http://www.tpc.org/tpc_documents_current_versions/pdf/tpch2.17.1.pdf.

[9] Whetlab. http://www.whetlab.com/.

[10] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing Data-parallel Computing. NSDI, 2012.

[11] E. Anderson. HPL-SSP-2001-4: Simple table-based modeling of storage devices, 2001.

[12] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. Automatic exploration of datacenter performance regimes. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*. ACM, 2009.

[13] E. Brochu, V. M. Cora, and N. De Freitas. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[14] V. Dalibard. A framework to build bespoke autotuners with structured Bayesian optimisation. Technical report, University of Cambridge, Computer Laboratory, 2017.

[15] C. Delimitrou and C. Kozyrakis. QoS-Aware Scheduling in Heterogeneous Datacenters with Paragon. *ACM Trans. Comput. Syst.*, 2013.

[16] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. ASPLOS. ACM, 2014.

[17] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, 2012.

[18] J. R. Gardner, M. J. Kusner, Z. E. Xu, K. Q. Weinberger, and J. Cunningham. Bayesian Optimization with Inequality Constraints. In *International Conference on Machine Learning*, 2014.

[19] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011.

[20] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A Self-tuning System for Big Data Analytics. In *Conference on Innovative Data Systems Research*, 2011.

[21] Y. Jiang, L. Ravindranath, S. Nath, and R. Govindan. WebPerf: Evaluating what-if scenarios for cloud-hosted web applications. In *SIGCOMM*, 2016.

[22] D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 1998.

[23] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: comparing public cloud providers. In *SIGCOMM*, 2010.

[24] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, and N. Fuller. MrOnline: MapReduce online performance tuning. In *International Symposium on High-performance Parallel and Distributed Computing*, 2014.

[25] I. Loshchilov and F. Hutter. CMA-ES for Hyperparameter Optimization of Deep Neural Networks. *arXiv preprint arXiv:1604.07269*, 2016.

[26] D. J. MacKay. Introduction to Gaussian processes. *NATO ASI Series F Computer and Systems Sciences*, 1998.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 2015.

[28] J. Mockus. *Bayesian approach to global optimization: theory and applications*. Springer Science & Business Media, 2012.

[29] O. O'Malley. Terabyte sort on apache hadoop. *Yahoo, available online at: `http://sortbenchmark.org/Yahoo-Hadoop.pdf`*, 2008.

[30] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. TritonSort: A balanced large-scale sorting system. In *NSDI*, 2011.

[31] C. E. Rasmussen. Gaussian processes for machine learning. MIT Press, 2006.

[32] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. Taking the human out of the loop: a review of Bayesian optimization. *Proceedings of the IEEE*, 2016.

[33] J. Snoek, H. Larochelle, and R. P. Adams. Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, 2012.

[34] I. M. Sobol. On quasi-monte carlo integrations. *Mathematics and Computers in Simulation*, 1998.

[35] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press Cambridge, 1998.

[36] V. N. Vapnik. *Statistical Learning Theory*. John Wiley & Sons, 1998.

[37] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI*, 2016.

[38] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*. ACM, 2011.

[39] G. Wang and T. Ng. The impact of virtualization on network performance of Amazon EC2 data center. In *INFOCOM*, 2010.

[40] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger. Storage device performance prediction with CART models. In *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*. IEEE, 2004.

[41] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.

[42] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *International Conference on World Wide Web*, 2004.

[43] T. Ye and S. Kalyanaraman. A recursive random search algorithm for large-scale network parameter configuration. *SIGMETRICS Perform. Eval. Rev.*, 2003.

[44] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot Topics in Cloud Computing*, 2010.

[45] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo. Automated Profiling and Resource Management of Pig Programs for Meeting Service Level Objectives. In *International Conference on Autonomic Computing*, 2012.

AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network

Daniel S. Berger[1], Ramesh K. Sitaraman[2], and Mor Harchol-Balter[3]
[1]*University of Kaiserslautern* [2]*UMass Amherst & Akamai Technologies* [3]*Carnegie Mellon University*

Abstract

Most major content providers use content delivery networks (CDNs) to serve web and video content to their users. A CDN is a large distributed system of servers that caches and delivers content to users. The first-level cache in a CDN server is the memory-resident Hot Object Cache (HOC). A major goal of a CDN is to maximize the object hit ratio (OHR) of its HOCs. But, the small size of the HOC, the huge variance in the requested object sizes, and the diversity of request patterns make this goal challenging.

We propose AdaptSize, the first *adaptive*, *size-aware* cache admission policy for HOCs that achieves a high OHR, even when object size distributions and request characteristics vary significantly over time. At the core of AdaptSize is a novel Markov cache model that seamlessly adapts the caching parameters to the changing request patterns. Using request traces from one of the largest CDNs in the world, we show that our implementation of AdaptSize achieves significantly higher OHR than widely-used production systems: 30-48% and 47-91% higher OHR than Nginx and Varnish, respectively. AdaptSize also achieves 33-46% higher OHR than state-of-the-art research systems. Further, AdaptSize is more robust to changing request patterns than the traditional tuning approach of hill climbing and shadow queues studied in other contexts.

1 Introduction

Content delivery networks (CDNs) [18] enhance performance by caching objects in servers close to users and rapidly delivering those objects to users. A large CDN, such as that operated by Akamai [57], serves trillions of user requests a day from 170,000+ servers located in 1500+ networks in 100+ countries around the world. CDNs carry the majority of today's Internet traffic and are expected to carry almost two thirds within five years [22].

A CDN server employs two levels of caching: a small but fast in-memory cache called the Hot Object Cache (HOC) and a large second-level Disk Cache (DC). Each requested object is first looked up in the HOC. If absent, it is looked up in the DC. If also absent there, the object is fetched over the WAN from the content provider's origin.

Serving requests from the HOC is much faster and more efficient than serving from the DC. Thus, the goal of a CDN is to maximize the *object hit ratio* (OHR), which is the fraction of requests served from the HOC.

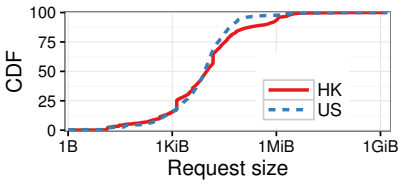


Figure 1: The cumulative distribution for object sizes in two Akamai production traces from Hong Kong and the US. Sizes vary by more than nine orders of magnitude.

In this paper, we show how to attain this goal in a robust, efficient and scalable manner (see Section 2).

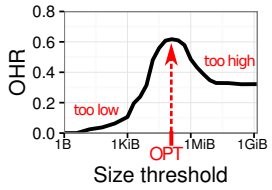
1.1 Why HOC cache management is hard

HOC cache management entails two types of decisions. First, the cache can decide whether or not to admit an object (cache admission). Second, the cache can decide which object to evict from the cache (cache eviction), if there is no space for a newly admitted object. While cache management is well studied in other contexts, HOC cache management poses the following new challenges.

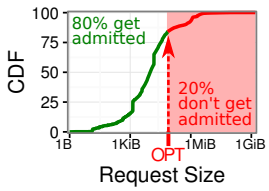
1) The HOC is subject to extreme variability in request patterns and object sizes. CDNs serve multiple traffic classes using a *shared* server infrastructure. Such classes include web sites, videos, and interactive applications from thousands of content providers, each class with its own distinctive object size distributions and request patterns [57]. Figure 1 shows the object size distribution of requests served by two Akamai production servers (one in the US, the other in Hong Kong). We find that object sizes span more than nine orders of magnitude, and that the largest objects are often of the same order of magnitude as the HOC size itself. This extreme variability underscores the need for cache admission, as placing one large object in the HOC can result in the eviction of many small ones, which can severely degrade the OHR (Section 3).

2) Prior academic research is largely inapplicable as it focuses on caching objects of similar sizes. While the academic literature on caching policies is extensive, it focuses on situations where objects are of the same size. Further, prior work almost exclusively focuses on *eviction policies*, i.e., *all* requested objects are *admitted* to the cache and space is only managed by eviction (Section 7). Thus, there is little emphasis on cache admission in the prior literature, even as we show that cache admission is key in our context (Section 3).

3) Production systems implement a cache admission scheme with a static threshold that performs sub-

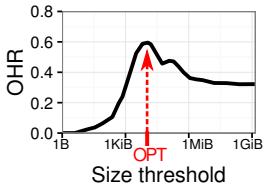


(a) OHR versus threshold.

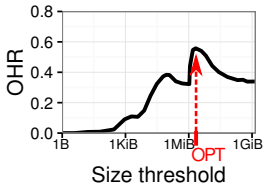


(b) Request size distribution.

Figure 2: Experimental results with different size thresholds. (a) A OHR-vs-threshold curve shows that the Object Hit Ratio (OHR) is highly sensitive to the size threshold, and that the optimal threshold (red arrow) can significantly improve the OHR. (b) The optimal threshold admits the requested object for only 80% of the requests.



(a) Morning: web traffic.



(b) Evening: web/video mix.

Figure 3: The optimal size threshold changes significantly over time. (a) In the morning hours, small objects (e.g., news items) are more popular, which requires a small size threshold of a few tens of KiBs. (b) In the evening hours, web traffic gets mixed with video traffic, which requires a size threshold of a few MiBs.

optimally for CDN workloads. In contrast to much of the academic research, production systems recognize the fact that *not all* objects can be admitted into the HOC. A common approach is to define a static size threshold and to only admit objects with size below this threshold. Figure 2a shows how OHR is affected by the size threshold for a production CDN workload. While the optimal threshold (OPT) almost doubles the OHR compared to admitting all objects, conservative thresholds that are too high lead to marginal gains, and the OHR quickly drops to zero for aggressive thresholds that are too low.

Unfortunately, the "best" threshold changes significantly over time. Figures 3a and 3b show the OHR as a function of the size threshold at two different times of the day. Note that the optimal thresholds can vary by as much as two orders of magnitude during a day. Since no prior method exists for dynamically tuning such a threshold, companies have resorted to either setting the size admission threshold conservatively high, or (more commonly) not using size-aware admission at all [67, 54, 21].

4) Simple strategies for dynamically tuning cache admission parameters do not work well. While it may seem that simple tuning approaches can be used to adapt the size threshold parameter over time, this turns out to be a non-trivial problem. This is probably why size-aware admission is not used effectively in practice. In Section 3, we consider common tuning approaches such as hill climbing with shadow caches, or using a threshold that is a fixed function of a request size percentile (e.g., the 80-th percentile as in Figure 2b). We also consider using probabilistic size-aware admission, where small sizes are "more likely" to be admitted, and large files are "less likely" to be admitted. We find that none of these approaches is sufficiently robust to traffic mix changes that occur in daily operation due in part to the CDN's global load balancer.

1.2 Our contributions

We propose AdaptSize, a lightweight and near-optimal tuning method for size-aware cache admission. AdaptSize is based on a novel statistical representation of the cache using a Markov model. This model is unique in that it incorporates all correlations between object sizes and current request rates (prior caching models assumed unit-sized objects). This model is analytically tractable, allowing us to quickly find the optimal size-aware admission policy and repeat this process at short intervals.

We have implemented AdaptSize within the Varnish production caching system[1] Varnish is known as a high-throughput system that makes extensive use of concurrency [76, 43, 42] and is used by several prominent content providers and CDNs, including Wikipedia, Facebook, and Twitter. Through evaluations of AdaptSize on production request traces from one of the world's largest CDNs, Akamai, we observe the following key features.

1. AdaptSize improves the OHR by 47-91% over an unmodified Varnish system, and by 30-48% over an offline-tuned version of the Nginx caching system (Figure 4 and Section 6.1). Varnish and Nginx are used by almost 80% of the top 5000 websites, which we determined by crawling Alexa's top sites list [74].
2. In addition to improving upon production systems, AdaptSize also improves the OHR by 33-46% over state-of-the-art research caching systems (Figure 5 and Section 6.2).
3. Compared to other tuning methods, such as the classical hill climbing technique using shadow queues, AdaptSize improves the OHR on average by 15-20% and in some cases by more than 100% (Figure 6 and Section 6.3). In particular, we found classical tuning methods can get "stuck" at poor local optima that are avoided by AdaptSize's model-based optimization.
4. We compare AdaptSize with SIZE-OPT, which tunes the size threshold parameter using a priori knowledge of the next one million requests. AdaptSize stays within 90% of the OHR of SIZE-OPT in the median across all experiments and is never worse than 80% of the OHR of SIZE-OPT (Sections 6.1

[1]The source code of AdaptSize and installation instructions are available at `https://github.com/dasebe/AdaptSize` .

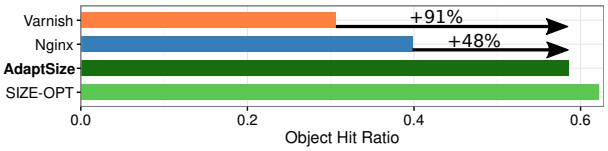


Figure 4: Comparison of AdaptSize's implementation to the Varnish and Nginx production systems. We also show the SIZE-OPT policy which has future knowledge and uses this to set the optimal size threshold at every moment in time. AdaptSize improves the OHR by 48-91% over the production systems and also achieves 95% of the OHR of SIZE-OPT. These results are for the US trace and a typical HOC size (details in Section 5.1).

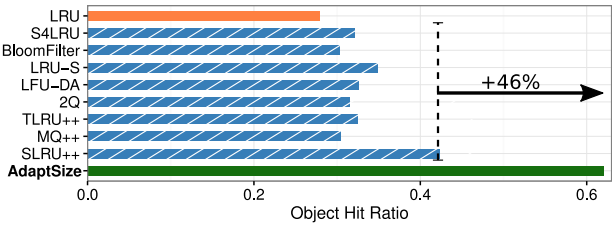


Figure 5: Comparison of AdaptSize to state-of-the-art research caching systems. Most of these use sophisticated admission and eviction policies that combine recency and frequency (striped blue bars). AdaptSize improves the OHR by 46% over the next best system. Policies annotated by "++" are actually optimistic, because we offline-tuned their parameters to the trace. These results are for the US trace and a HOC size 1.2 GiB.

and 6.3) – even when subjected to extreme changes in the request traffic.

5. In addition to improving the OHR, AdaptSize also reduces request latencies by 43% in the median, and by more than 30% at the 90-th and 99.9-th percentile. AdaptSize is able to maintain the high throughput of Varnish without adding (concurrent) synchronization overheads, and reduces the disk utilization of the second-level cache by 20% (Section 6.4).

Roadmap. The rest of this paper is structured as follows. Sections 2 and 3 discuss our goals and our rationale in designing AdaptSize. Section 4 details AdaptSize's design and implementation. Sections 5 and 6 describe our setup and experimental results, respectively. Section 7 reviews related work. We conclude in Section 8.

2 HOC Design Goals

In designing AdaptSize, we seek to maximize the OHR, while maintaining a robust and scalable HOC and avoiding adverse side-effects on second-level caches.

Maximizing the OHR. The HOC's primary design objective is user performance, which it optimizes by providing fast responses for as many requests as possible. A natural way to measure this objective is the *object hit ratio* (OHR), which gives equal weight to all user requests.

While our evaluations are based on production CDN servers without SSDs, HOCs are also key to hybrid CDN servers that typically have both hard disks and SSDs. This is because HOCs are more CPU efficient and can serve traffic at higher rates. Further, HOCs offload requests from the SSDs that are often i/o bound. HOCs are used in SSD-based servers at Akamai, and also at Fastly [54] and Wikipedia [67]. These production deployments seek to maximize the OHR of the HOC, which is the main performance metric used throughout this paper.

There are other cache performance metrics that are less relevant to the HOC. For example, the much larger DC focuses on the byte hit rate (BHR) that is the fraction of *bytes* that are served from the cache [71]. The HOC has little impact on the BHR as it is typically three orders of magnitude smaller than the DC.

Robustness against changing request patterns. A HOC is subjected to a variety of traffic changes each day. For example, web content popularity changes during the day (e.g., news in the morning vs. video at night), which includes rapid changes due to flash crowds. Another source of traffic changes is the sharing of the server infrastructure between traffic classes. Such classes include web sites, videos, software downloads, and interactive applications from thousands of content providers [57]. As a shared infrastructure is more cost effective, a CDN server typically serves a mix of traffic classes. Due to load balancing decisions, this mix can change abruptly. This poses a particular challenge as each traffic class has its own distinctive request and object size distribution statistics: large objects can be unpopular during one hour and popular during the next. A HOC admission policy must be able to rapidly adapt to all these changing request patterns in order to achieve consistently high OHRs.

Low overhead and high concurrency. As the first caching level in a CDN, the HOC needs to both respond quickly to requests and deliver high throughput. This requires that the admission and eviction policies have a small processing overhead, i.e., a constant time complexity per request (see the $O(1)$ policies in Table 2 in Section 7), and that they have concurrent implementations (see the corresponding column in Table 2).

No negative side-effects. While the HOC achieves a high OHR and fast responses, it must not impede the performance of the overall CDN server. Specifically, changes to the HOC must not negatively affect the BHR and disk utilization of the DC.

3 Rationale for AdaptSize

The goal of this section is to answer why the HOC needs size-aware admission, why such an admission policy

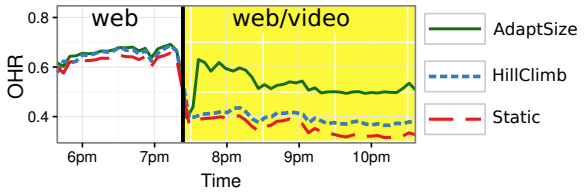


Figure 6: Comparison of AdaptSize, threshold tuning via hill climbing and shadow caches (HillClimb), and a static size threshold (Static) under a traffic mix change from only web to mixed web/video traffic. While AdaptSize quickly adapts to the new traffic mix, HillClimb gets stuck in a suboptimal configuration, and Static (by definition) does not adapt. AdaptSize improves the OHR by 20% over HillClimb and by 25% over Static on this trace.

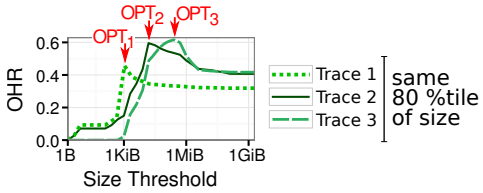


Figure 7: Experimental results showing that setting the size threshold to a fixed function does not work. All three traces shown here have the same 80-th size percentile, but their optimal thresholds differ by two orders of magnitude.

needs to be adaptively tuned, and why a new approach to parameter tuning is needed.

3.1 Why HOCs need size-aware admission

We start with a toy example. Imagine that there are only two types of objects: 9999 small objects of size 100 KiB (say, web pages) and 1 large object of size 500 MiB (say, a software download). Further, assume that all objects are equally popular and requested forever in round-robin order. Suppose that our HOC has a capacity of 1 GiB.

A HOC that does not use admission control cannot achieve an OHR above 0.5. Every time the large object is requested, it pushes out ≈5000 small objects. It does not matter which objects are evicted: when the evicted objects are requested, they cannot contribute to the OHR.

An obvious solution for this toy example is to control admissions via a size threshold. If the HOC admits only objects with a size at most 100 KiB, then it can achieve an OHR of 0.9999 as all small objects stay in the cache.

This toy example is illustrative of what happens under real production traffic. We observe from Figure 1 that approximately 5% of objects have a size bigger than 1 MiB. Every time a cache admits a 1 MiB object, it needs to evict space equivalent to one thousand 1 KiB objects, which make up about 15% of requests. Again, those evicted objects will not be able to contribute to the OHR. A well-designed cache admission algorithm could help avoid such evictions that have a large impact on OHR.

3.2 Why we need a new tuning method

The key question when implementing size-aware admission is picking its parameters. Figure 2 shows that a static size threshold is inadequate. Next, we explore three canonical approaches for tuning a size threshold. These approaches are well-known in prior literature and have been applied in other contexts (unrelated to the tuning of size thresholds). However, we show that these known approaches are deficient in our context, motivating the need for AdaptSize's new tuning mechanism.

Tuning based on request size percentiles. A common approach used in many contexts (e.g., capacity provisioning) is to derive the required parameter as some function of the request size distribution and arrival rate. A simple way of using this approach in our context is to set the size threshold for cache admission to be a fixed percentile of the object size distribution. However, for production CDN traces, there is no fixed relationship between the percentiles of the object size distribution and optimal size threshold that maximizes the OHR. In Figure 2, the optimal size threshold lands on the 80-th percentile request size. However, in Figure 7, note that all three traces have the same 80-th percentile but very different optimal thresholds. In fact, we found many examples of multiple traces that agree on *all* size percentiles and yet have *different* optimal size thresholds. The reason is that for maximizing OHR it matters whether the number of requests seen for a specific object size come from one (very popular) object or from many (unpopular) objects. This information is not captured by the request size distribution.

Tuning via hill climbing and shadow caches. A common tool for the tuning of caching parameters is the use of shadow caches. For example, in the seminal paper on ARC [53], the authors tune their eviction policy to have the optimal balance between recency and frequency by using a shadow cache (we discuss other related work using shadow caches in Section 7.2). A shadow cache is a simulation which is run in real time simultaneously with the main (implemented) cache, but using a different parameter value than the main cache. Hill climbing then adapts the parameter by comparing the hit ratio achieved by the shadow cache to that of the main cache (or another shadow cache). In theory, we could exploit the same idea to set our size-aware admission threshold. Unfortunately, when we tried this, we found that the OHR-vs-threshold curves are not concave and that they can have several local optima, in which the hill climbing gets frequently stuck. Figure 3b shows such an example, in which the local optima result from mixed traffic (web and video). As a consequence, we will demonstrate experimentally in Section 6.3 that hill climbing is suboptimal. AdaptSize achieves an OHR that is 29% higher than hill climbing on

average and 75% higher in some cases. We tried adding more shadow caches, and also randomizing the evaluated parameters, but could not find a robust variant that consistently optimized the OHR across multiple traces[2].

In conclusion, our extensive experiments show that tuning methods like shadow caches with hill climbing are simply not robust enough for the problem of size-aware admission with CDN traffic.

Avoiding tuning by using probabilistic admission. One might imagine that the difficulty in tuning the size threshold lies in the fact that we are limited to a single strict threshold. The vast literature on randomized algorithm suggests that probabilistic parameters are more robust than deterministic ones [55]. We attempted to apply this idea to size-aware tuning by considering probabilistic admission policies, which "favor the smalls" by admitting them with high probability, whereas large objects are admitted with low probability. We chose a probabilistic function that is exponentially decreasing in the object size ($e^{-size/c}$). Unfortunately, the parameterization of the exponential curve (the c) matters a lot – and it's just as hard to find this c parameter as it is to find the optimal size threshold. Furthermore, the best exponential curve (the best c) changes over time. In addition to exponentially decreasing probabilities, we also tried inversely proportional, linear, and log-linear variants. Unfortunately, none of these variants resolves the problem that there is at least one parameter without an obvious way how to choose it.

In conclusion, even randomized admission control requires the tuning of some parameter.

4 The AdaptSize Caching System

AdaptSize admits objects with probability $e^{-size/c}$ and evicts objects using a concurrent variant of LRU [43]. Observe that the function $e^{-size/c}$ is biased in favor of admitting small sizes with higher *probability*.

Why a probabilistic admission function? The simplest size-based admission policy is a deterministic threshold c where only objects with a size $< c$ are admitted. A probabilistic admission function, like $e^{-size/c}$, is more flexible: objects greater than c retain a low but non-zero admission probability, which results in eventual admission for popular objects (but not for unpopular ones). In our experiments $e^{-size/c}$ consistently achieves a 10% higher OHR than the best deterministic threshold.

What parameter c does AdaptSize use in the $e^{-size/c}$ function? AdaptSize's tuning policy recomputes the optimal c every Δ requests. A natural approach is to use hill-climbing with shadow caches to determine the optimal c parameter. Unfortunately, that leads to a myopic view in that only a *local* neighborhood of the current c can be searched. This leads to sub-optimal results, given the non-convexities present in the OHR-vs-c curve (Figure 3). By contrast, we derive a full Markov chain model of the cache. This model allows AdaptSize to view the entire OHR-vs-c curve and perform a *global* search for the optimal c. The challenge of the Markov model approach is in devising an algorithm for finding the solution quickly and in incorporating that algorithm into a production system.

[2]While there are many complicated variants of shadow-cache search algorithms, they all rely on a fundamental assumption of stationarity, which does not need to apply to web traffic.

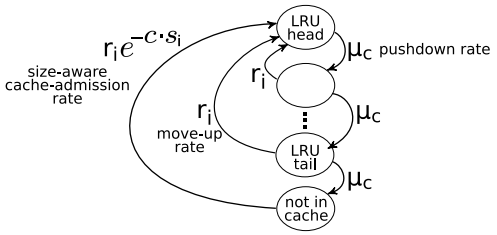


Figure 8: AdaptSize's Markov chain model for object i represents i's position in the LRU list and the possibility that the object is out of the cache. Each object is represented by a separate Markov chain, but all Markov chains are connected by the common "pushdown" rate μ_c. Solving these models yields the OHR as a function of c.

In the following, we describe the derivation of AdaptSize's Markov model (Section 4.1), and how we incorporate AdaptSize into a production system (Section 4.2).

4.1 AdaptSize's Markov chain tuning model

To find the optimal c, AdaptSize uses a novel Markov chain model, which is very different from that typically used for cache modeling. Traditionally, people have modeled the entire state of the cache, tracking all objects in the cache and their ordering in the LRU list [44, 30, 15, 52, 10, 33, 25, 24, 19, 68]. While this is 100% accurate, it also becomes completely infeasible when the number of objects is high, because of a combinatorical state space explosion.

AdaptSize instead creates a *separate* Markov chain for each object (cf. Figure 8). Each object's chain tracks its position in the LRU list (if the object is in the cache), as well as a state for the possibility that the object is out of the cache. Using an individual Markov chain greatly reduces the model complexity, which now scales *linearly* with the number of objects, rather than *exponentially* in the number of objects.

AdaptSize's Markov chain. Figure 8 shows the Markov chain for the i^{th} object. The chain has two important parameters. The first is the rate at which object i is moved *up* to the head of the LRU list, due to accesses to the object. We get the "move up" rate, r_i, by collecting aggregate statistics for object i during the previous Δ time interval. The second parameter is the average rate at which object i is pushed *down* the LRU list. The "pushdown" rate, μ_c, depends on the rate with which any object is moved to the top of the LRU list (due to a hit,

or after cache admission). As it does not matter which object is moved to the top, μ_c is approximately the same for all objects. So, we consider a single "pushdown" rate for all objects. We calculate μ_c by solving an equation that takes all objects into account, and thus captures the interactions between all the objects[3]. Specifically, we find μ_c by solving an equation that says that the expected size of all cached objects can't exceed the capacity K that is actually available to the cache:

$$\sum_{i=1}^{N} \mathbb{P}[\text{object } i \text{ in cache}]\ s_i = K\ . \qquad (1)$$

Here, N is the number of all objects observed over the previous Δ interval, and s_i is the size of object i. Note that $\mathbb{P}[\text{object } i \text{ in cache}]$ is a monotonic function in terms of μ_c, which leads to a unique solution.

Our new model enables us to find $\mathbb{P}[\text{object } i \text{ in cache}]$ as a function of c by solving for the limiting probabilities of all "in" states in Figure 8. Appendix A shows how this is done. We obtain a function of c in closed form.

Theorem 1 *(proof in Appendix A)*

$$\mathbb{P}[\textit{object } i \textit{ in cache}] = \frac{(e^{r_i/\mu_c} - 1) \cdot e^{-c \cdot s_i}}{1 + (e^{r_i/\mu_c} - 1) \cdot e^{-c \cdot s_i}}$$

Note that the size admission parameter c affects both the admission probability ($e^{-s_i/c}$) and the pushdown rate (μ_c). For example, a lower c results in fewer admissions, which results in fewer evictions, and in a smaller pushdown rate.

The OHR as a function of c. Theorem 1 and Equation (1) yield the OHR by observing that the expected number of hits of object i equals r_i (i's average request rate) times the long-term probability that i is in the cache. The OHR predicted for the threshold parameter c is then simply the ratio of expected hits to requests:

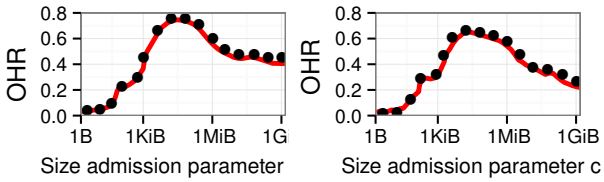
$$OHR(c) = \frac{\sum_{i=1}^{N} r_i \mathbb{P}[\text{object } i \text{ in cache}]}{\sum_{i=1}^{N} r_i}\ .$$

If we consider a discretized range of c values, we can now compute the OHR for each c in the range which gives us a "curve" of OHR-vs-c (similar to the curves in Figure 9).

Global search for the optimal c. Every Δ steps, we derive the OHR-vs-c curve using our Markov model. We search this curve for the c that maximizes the OHR using a standard global search method for non-concave functions [64]. This c is then used for the next Δ steps.

Accuracy of AdaptSize's model. Our Markov chain relies on several simplifying assumptions that can potentially impact the accuracy of the OHR predictions. Figure 9 shows that AdaptSize's OHR equation matches experimental results across the whole range of the threshold parameter c on two typical traces of length Δ. In addition, we continuously compared AdaptSize's model to measurements during our experiments (Section 6). AdaptSize is very accurate with an average error of about 1%.

[3]Mean-field theory [45] provides analytical justification for why it is reasonable to assume a single average pushdown rate, when there are thousands of objects (as in our case).



(a) HK trace (b) US trace

Figure 9: AdaptSize's Markov model predicts the OHR sensitivity curve (red solid line). This is very accurate when compared to the actual OHR (black dots) that results when that threshold is chosen. Each experiment involves a portion of the production trace of length $\Delta = 250K$.

4.2 Integration with a production system

We implemented AdaptSize on top of Varnish [76, 32], a production caching system, by modifying the miss request path. On a cache miss, Varnish accesses the second-level cache to retrieve the object, and places it in its HOC. With AdaptSize, the probabilistic admission decision is executed, which is evaluated independently for all cache threads and adds a constant number of instructions to the request path. If the object is not admitted, it is served from Varnish's transient memory.

Our implementation uses a parameter Δ which is the size of the window of requests over which our Markov model for tuning is computed. In addition to statistics from the current window, we also incorporate the statistical history from prior windows via exponential smoothing, which makes AdaptSize more robust and largely insensitive to Δ on both of our production traces. In our experiments, we choose Δ=250K requests (about 5-10 mins on average), which allows AdaptSize to react quickly to changes in the request traffic.

Lock-free statistics collection. A key problem in implementing AdaptSize lies in efficient statistics collection for the tuning model. Gathering request statistics can add significant overhead to concurrent caching designs [69]. Varnish and AdaptSize use thousands of threads in our experiments, so centralized request counters would cause high lock contention. In fact, we find that Varnish's throughput bottleneck is lock contention for the few remaining synchronization points (e.g., [43]).

Instead of a central request counter, AdaptSize hooks into the internal data structure of the cache threads. Each cache thread keeps debugging information in a concurrent ring buffer, to which all events are simply appended (overwriting old events after some time). AdaptSize's statistics collection frequently scans this ring buffer (read only) and does not require any synchronization.

Robust and efficient model evaluation. The OHR prediction in our statistical model involves two more implementation challenges. The first challenge lies in efficiently solving equation (1). We achieve a constant time

	HK trace	US trace
Total Requests	450 million	440 million
Total Bytes	157.5 TiB	152.3 TiB
Unique Objects	25 million	55 million
Unique Bytes	14.7 TiB	8.9 TiB
Start Date	Jan 29, 2015	Jul 15, 2015
End Date	Feb 06, 2015	Jul 20, 2015

Table 1: Basic information about our web traces.

overhead by using a fixed-point solver [26]. The second challenge is due to the exponential function in the Theorem 1. The value of the exponential function outgrows even 128-bit float number representations. We solve this problem by using an accurate and efficient approximation for the exponential function using a Padé approximant [63] that only uses simple float operations which are compatible with SSE/AVX vectorization, speeding up the model evaluation by about 10-50$\times$ in our experiments.

5 Evaluation Methodology

We evaluate AdaptSize using *both* trace-based simulations (Section 5.2) and a Varnish-based implementation (Section 5.3) running on our experimental testbed. For both these approaches, the request load is derived from traces from Akamai's production CDN servers (Section 5.1).

5.1 Production CDN request traces

We collected request traces from two production CDN servers in Akamai's global network. Table 1 summarizes the main characteristics of the two traces. Our first trace is from urban Hong Kong (**HK trace**). Our second trace is from rural Tennessee, in the US, (**US trace**). Both span multiple consecutive days, with over 440 million requests per trace during the months of February and July 2015. Both production servers use a HOC of size 1.2 GiB and several hard disks as second-level caches. They serve a traffic mix of several thousand popular web sites, which represents a typical cross section of the web (news, social networks, downloads, ecommerce, etc.) with highly variable object sizes. Some content providers split very large objects (e.g., videos) into smaller (e.g., 2 MiB) chunks. The chunking approach is accurately represented in our request traces. For example, the cumulative distribution function shown in Figure 1 shows a noticeable jump around the popular 2 MiB chunk size.

5.2 Trace-based simulator

We implemented a cache simulator in C++ that incorporates AdaptSize and several state-of-the-art research caching policies. The simulator is a single-threaded implementation of the admission and eviction policies and performs the appropriate cache actions when it is fed the CDN request traces. Objects are only stored via their ids and the HOC size is enforced by a simple check on the sum of bytes currently stored. While actual caching systems (such as Varnish [43, 42]) use multi-threaded concurrent implementations, our simulator provides a good approximation of the OHR when compared with our prototype implementations that we describe next.

5.3 Prototype Evaluation Testbed

Our implementation testbed is a dedicated (university) data center consisting of a client server, an origin server, and a CDN server that incorporates the HOC. We use FUJITSU CX250 HPC servers, which run RHEL 6.5, kernel 2.6.32 and gcc 4.4.7 on two Intel E5-2670 CPUs with 32 GiB RAM and an IB QDR networking interface.

In our evaluation, the HOC on our CDN server is either running Nginx, Varnish, or AdaptSize. Recall that we implemented AdaptSize by adding it to Varnish[4] as described in Section 4.2. We use Nginx 1.9.12 (February 2016) with its build-in frequency-based admission policy. This policy relies on one parameter: how many requests need to be seen for an object before being admitted to the cache. We use an optimized version of Nginx, since we have tuned its parameter offline for both traces. We use Varnish 4.1.2 (March 2016) with its default configuration that does not use an admission policy.

The experiments in Section 6.1, 6.2, and 6.3 focus on the HOC and do not use a DC. The DC in Section 6.1 uses Varnish in a configuration similar to that of the Wikimedia Foundation's CDN [67]. We use four equal dedicated 1 TB WD-RE3 7200 RPM 32 MiB-Cache hard disks attached via a Dell 6 Gb/s SAS Host Bus Adapter Card in raw mode (RAID disabled).

The client fetches content specified in the request trace from the CDN server using libcurl. The request trace is continuously read into a global queue, which is distributed to worker threads (client threads). Each client thread continually requests objects in a closed-loop fashion. We use up to 200 such threads and verified that the number of client threads has a negligible impact on the OHR.

If the CDN server does not have the requested content, it is fetched from the origin server. Our origin server is implemented in FastCGI. As it is infeasible to store all trace objects (23 TB total) on the origin server, our implementation creates objects with the correct size on the fly before sending them over the network. In order to stress test our caching implementation, the origin server is highly multi-threaded and intentionally never the bottleneck.

6 Empirical Evaluation

This section presents our empirical evaluation of AdaptSize. We divide our evaluation into three parts. In Section 6.1, we compare AdaptSize with production caching systems, as well as with an *offline* caching system called SIZE-OPT that continuously optimizes OHR with knowledge of future requests. While SIZE-OPT is not implementable in practice. it provides an upper bound on the

[4]We refer to AdaptSize incorporated into Varnish as "AdaptSize" and Varnish without modifications as "Varnish".

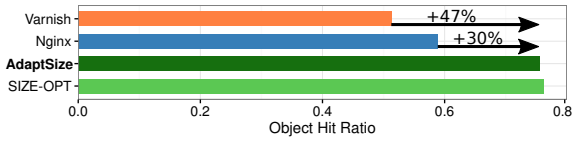


Figure 10: Comparison of AdaptSize's implementation to the Varnish and Nginx production systems and SIZE-OPT. AdaptSize improves the OHR by 30-47% over the production systems and also achieves 99% of the OHR of SIZE-OPT. These results are for the HK trace; corresponding results for the US trace are shown in Figure 4.

achievable OHR to which AdaptSize can be compared. In Section 6.2, we compare AdaptSize with research caching systems that use more elaborate eviction and admission policies. In Section 6.3, we evaluate the robustness of AdaptSize by emulating both randomized and adversarial traffic mix changes. In Section 6.4, we evaluate the side-effects of AdaptSize on the overall CDN server.

6.1 Comparison with production systems

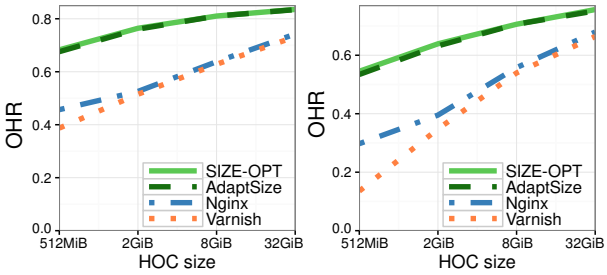
We use our experimental testbed outlined in Section 5.3 and answer four basic questions about AdaptSize.

What is AdaptSize's OHR improvement over production systems? Quick answer: *AdaptSize improves the OHR by 47-91% over Varnish and by 30-48% over Nginx*. We compare the OHR of AdaptSize to Nginx and Varnish using the 1.2 GiB HOC configuration from the corresponding Akamai production servers (Section 5.1). For the HK trace (Figure 10), we find that AdaptSize improves over Nginx by 30% and over Varnish by 47%. For the US trace (Figure 4), the improvement increases to 48% over Nginx and 91% over Varnish.

The difference in the improvement over the two traces stems from the fact that the US trace contains 55 million unique objects as compared to only 25 million unique objects in the HK trace. We further find that AdaptSize improves the OHR variability (the coefficient of variation) by 1.9× on the HK trace and by 3.8× on the US trace (compared to Nginx and Varnish).

How does AdaptSize compare with SIZE-OPT? Quick answer: *for the typical HOC size, AdaptSize achieves an OHR within 95% of SIZE-OPT*. We benchmark AdaptSize against the SIZE-OPT policy, which tunes the threshold parameter c using a priori knowledge of the next one million requests. Figures 4 and 10 show that AdaptSize is within 95% of SIZE-OPT on the US trace, and within 99% of SIZE-OPT on the HK trace, respectively.

How much is AdaptSize's performance affected by the HOC size? Quick answer: *AdaptSize's improvement over production caching systems becomes greater for smaller HOC sizes, and decreases for larger HOC sizes*. We consider the OHR when scaling the HOC size between 512 MiB and 32 GiB under the production server traffic



(a) HK trace (b) US trace

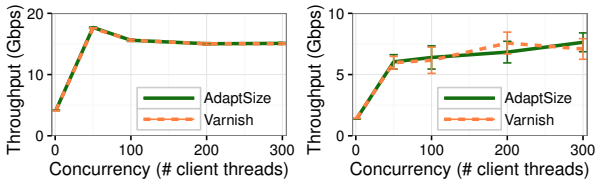
Figure 11: Comparison of AdaptSize to SIZE-OPT, Varnish, and Nginx when scaling the HOC size under the production server traffic of two 1.2 GiB HOCs. AdaptSize always stays close to SIZE-OPT and significantly improves the OHR for all HOC sizes.

of a 1.2 GiB HOC. Figures 11a and 11b shows that the performance of AdaptSize is close to SIZE-OPT for all HOC sizes. The improvement of AdaptSize over Nginx and Varnish is most pronounced for HOC sizes close to the original configuration. As the HOC size increases, the OHR of all caching systems improves, since the HOC can store more objects. This leads to a smaller relative improvement of AdaptSize for a HOC size of 32 GiB: 10-12% over Nginx and 13-14% over Varnish.

How much is AdaptSize's performance affected when jointly scaling up HOC size and traffic rate? Quick answer: *AdaptSize's improvement over production caching systems remains constant for larger HOC sizes*. We consider the OHR when jointly scaling the HOC size and the traffic rate by up 128x (153 GiB HOC size). This is done by splitting a production trace into 128 non-overlapping segments and replaying all 128 segments concurrently. We find that the OHR remains approximately constant as we scale up the system, and that AdaptSize achieves similar OHR improvements as under the original 1.2 GiB HOC configuration.

What about AdaptSize's overhead? Quick answer: *AdaptSize's throughput is comparable to existing production systems and AdaptSize's memory overhead is reasonably small*. AdaptSize is build on top of Varnish, which focuses on high concurrency and simplicity. In Figure 12, we compare the throughput (bytes per second of satisfied requests) of AdaptSize to an unmodified Varnish system. We use two micro experiments. The first benchmarks the hit request path (100% OHR scenario), to verify that there is indeed no overhead for cache hits (see section 4.2). The second benchmarks the miss request path (0% OHR scenario), to assess the worst-case overhead due to the admission decision.

We replay one million requests and configure different concurrency levels via the number of client threads. Note that a client thread does not represent an individual user (Section 5.3). The results are based on 50 repetitions.



(a) 100% OHR scenario. (b) 0% OHR scenario.

Figure 12: Comparison of the throughput of AdaptSize and Varnish in micro experiments with (a) 100% OHR and (b) 0% OHR. Scenario (a) stress tests the hit request path and shows that there is no difference between AdaptSize and Varnish. Scenario (b) stress tests the miss request path (every request requires an admission decision) and shows that the throughput of AdaptSize and Varnish is very close (within confidence intervals).

Figure 12a shows that the application throughput of AdaptSize and Varnish are indistinguishable in the 100% OHR scenario. Both systems achieve a peak throughput of 17.5 Gb/s for 50 clients threads. Due to lock contention, the throughput of both systems decreases to around 15 Gb/s for 100-300 clients threads. Figure 12b shows that the application throughput of both systems in the 0% OHR scenario is very close, and always within the 95% confidence interval.

The memory overhead of AdaptSize is small. The memory overhead comes from the request statistics needed for AdaptSize's tuning model. Each entry in this list describes one object (size, request count, hash), which requires less than 40 bytes. The maximum length of this list, across all experiments, is 1.5 million objects (58 MiB), which also agrees with the memory high water mark (VmHWM) reported by the Kernel for AdaptSize's tuning process.

6.2 Comparison with research systems

We have seen that AdaptSize performs very well against production systems. We now ask the following.

How does AdaptSize compare with research caching systems, which involve more sophisticated admission and eviction policies? Quick answer: *AdaptSize improves by 33-46% over state-of-the-art research caching system*. We use the simulation evaluation setup explained in Section 5.2 with eight systems from Table 2, which are selected with the criteria of having an efficient constant-time implementation. Four of the eight systems use a recency and frequency trade-off with fixed weights between recency and frequency. Another three systems (ending with "++") use sophisticated recency and frequency trade-offs with variable weights, which we hand-tuned to our traces to create optimistic variants[5]. The

[5]There are self-tuning variants of recency-frequency trade-offs such as ARC [53]. Unfortunately, we could not test ARC itself, because its learning rule relies on the assumption of unit-sized object sizes.

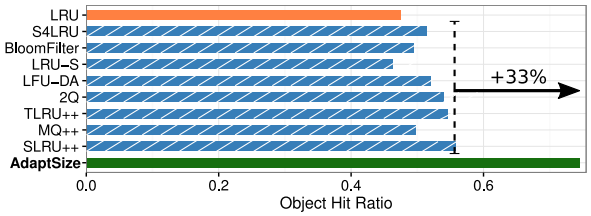


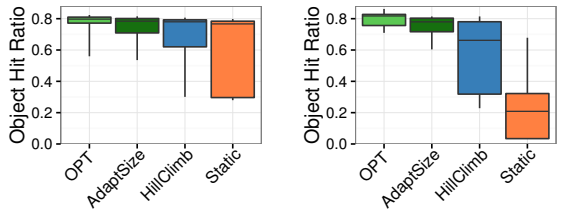
Figure 13: Comparison of AdaptSize to state-of-the-art research caching systems. Most of these are sophisticated admission and eviction policies that combine recency and frequency (striped blue bars). LRU-S is the only system – besides AdaptSize – that incorporates size. AdaptSize improves the OHR by 33% over the next best system. Policies annotated by "++" are actually optimistic, because we offline-tuned their parameters to the trace. These results are for the HK trace; correspondings results for the US trace are shown in Figure 5.

remaining system is LRU-S [72], which uses size-aware eviction and admission with static parameters.

Figure 13 shows the simulation results for a HOC of size 1.2 GiB on the HK trace. We find that AdaptSize achieves a 33% higher OHR than the second best system, which is SLRU++. Figure 5 shows the simulation results for the US trace. AdaptSize achieves a 46% higher OHR than the second best system, which is again SLRU++. Note that SLRU's performance heavily relies on offline parameters as can be seen by the much smaller OHR of S4LRU, which is a static-parameter variant of SLRU. In contrast, AdaptSize achieves its superior performance without needing offline parameter optimization. In conclusion, we find that AdaptSize's policies outperform sophisticated eviction and admission policies, which do not depend on the object size.

6.3 Robustness of alternative tuning methods for size-aware admission

So far we have seen that AdaptSize significantly improves the OHR over caching systems without size-aware admission, including production caching systems (Section 6.1) and research caching systems (Section 6.2). We now focus on different cache tuning methods for the size-aware admission parameter c (see the beginning of Section 4). Specifically, we compare AdaptSize with hill climbing (**HillClimb**), based on shadow caches (cf. Section 3). HillClimb uses two shadow caches and we hand-optimized its parameters (interval of climbing steps, step size) on our production traces. We also compare to a static size threshold (**Static**), where the value of this static threshold is offline optimized on our production traces. We also compare to SIZE-OPT, which tunes c based on offline knowledge of the next one million requests. All four policies are implemented on Varnish using the setup explained in Section 5.3.



(a) Randomized. (b) Adversarial.

Figure 14: Comparison of cache tuning methods under traffic mix changes. We performed 50 randomized traffic mix changes (a), and 25 adversarial traffic mix changes (b). The boxes show the range of OHR from the 25-th to the 75-th percentile among the 25-50 experiments. The whiskers show the 5-th to the 95-th percentile.

We consider two scenarios: 1) *randomized traffic mix changes* and 2) *adversarial traffic mix changes*. A randomized traffic mix change involves a random selection of objects which abruptly become very popular (similar to a flash crowd event). An adversarial traffic mix change involves frequently changing the traffic mix between classes that require vastly different size-aware admission parameters (e.g., web, video, or download traffic). An example of an adversarial change is the case where objects larger than the previously-optimal threshold suddenly become very popular.

Is AdaptSize robust against randomized traffic mix changes? Quick answer: *AdaptSize performs within 95% of SIZE-OPT's OHR even for the worst 5% of experiments, whereas HillClimb and Static achieve only 47-54% of SIZE-OPT's OHR*. We create 50 different randomized traffic mix changes. Each experiment consists of two parts. The first part is five million requests long, and allows each tuning method to converge to a stable configuration. The second part is ten million requests long, and consists of 50% production-trace requests and 50% of very popular objects. The very popular objects consist of a random number of objects (between 200 and 1000), which are randomly sampled from the trace.

Figure 14a shows a boxplot of the OHR for each caching tuning method across the 50 experiments. The boxes indicate the 25-th and 75-th percentile, the whiskers indicate the 5-th and 95-th percentile. AdaptSize improves the OHR over HillClimb across every percentile, by 9% on average, and by more than 75% in five of the 50 experiments. AdaptSize improves the OHR over Static across every percentile, by 30% on average, and by more than 100% in five of the 50 experiments. Compared to SIZE-OPT, AdaptSize achieves 95% of the OHR for all percentiles.

Is AdaptSize robust against adversarial traffic mix changes? Quick answer: *AdaptSize performs within 81% of SIZE-OPT's OHR even for the worst 5% of experiments, whereas HillClimb and Static achieve only 5-15% of SIZE-OPT's OHR*. Our experiment consists of 25 traffic mix changes. Each traffic mix is three million requests long, and the optimal c parameter changes from 32-256 KiB to 1-2 MiB, then to 16-32 MiB, and back again.

Figure 14b shows a boxplot of the OHR for each caching tuning method across all 50 experiments. The boxes indicate the 25-th and 75-th percentile, the whiskers indicate the 5-th and 95-th percentile. AdaptSize improves the OHR over HillClimb across every percentile, by 29% on average, and by more than 75% in seven of the 25 experiments. AdaptSize improves the OHR over Static across every percentile, by almost 3x on average, and by more than 10x in eleven of the 25 experiments. Compared to SIZE-OPT, AdaptSize achieves 81% of the OHR for all percentiles.

6.4 Side effects of Size-Aware Admission

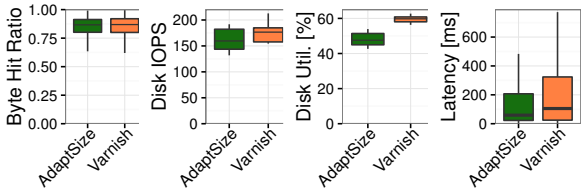
So far our evaluation has focused on AdaptSize's improvement with regard to the OHR. We evaluate AdaptSize's side-effects on the DC and on the client's request latency (cf. Section 2). Specifically, we compare AdaptSize to an unmodified Varnish system using the setup explained in Section 5.3. Network latencies are emulated using the Linux kernel (tc-netem). We set a 30ms round-trip latency between client and CDN server, and 100ms round-trip latency between CDN server and origin server. We answer the following three questions on the CDN server's performance.

How much does AdaptSize affect the BHR of the DC? Quick answer: *AdaptSize has a neutral effect on the BHR of the DC*. The DC's goal is to maximize the BHR, which is achieved by a very large DC capacity [49]. In fact, compared to the DC the HOC has less than on thousandth the capacity. Therefore, changes to the HOC have little effect on the DC's BHR.

In our experiment, we measure the DC's byte hit ratio (BHR) from the origin server. Figure 15a shows that there is no noticeable difference between the BHR under AdaptSize and under an unmodified Varnish.

Does AdaptSize increase the load of the DC's hard-disks? Quick answer: *No. In fact, AdaptSize reduces the average disk utilization by 20%*. With AdaptSize, the HOC admits fewer large objects, but caches many more small objects. The DC's request traffic therefore consists of more requests to large objects, and significantly fewer requests to small objects.

We measure the request size distribution at the DC and report the corresponding histogram in Figure 16. We observe that AdaptSize decreases the number of cache misses significantly for all object sizes below 256 KiB. For object sizes above 256 KiB, we observe a slight increase in the number of cache misses. Overall, we find that the DC has to serve 60% fewer requests with AdaptSize, but that the disks have to transfer a 30% higher



(a) BHR. (b) IOPS. (c) Utilization. (d) Latency.

Figure 15: Evaluation of AdaptSize's side effects across ten different sections of the US trace. AdaptSize has a neutral impact on the byte hit ratio, and leads to a 10% reduction in the median number of I/O operations going to the disk, and a 20% reduction in disk utilization.

byte volume. The average request size is also 4x larger with AdaptSize, which improves the sequentiality of disk access and thus makes the DC's disks more efficient.

To quantify the performance impact on the DC's hard-disks we use iostat [31]. Figure 15b shows that the average rate of I/O operations per second decreases by about 10%. Moreover, Figure 15c shows that AdaptSize reduces the disk's utilization (the fraction of time with busy periods) by more than 20%. We conclude that the increase in byte volume is more than offset by the fact that AdaptSize shields the DC from many small requests and also improves the sequentiality of requests served by the DC.

How much does AdaptSize reduce the request latency? Quick answer: *AdaptSize reduces the request latency across all percentiles by at least 30%.*

We measure the end-to-end request latency (time until completion of a request) from the client server. Figure 15d shows that AdaptSize reduces the median request latency by 43%, which is mostly achieved by the fast HOC answering a higher fraction of requests. The figure also shows significant reduction of tail latency, e.g., the 90-th and 99-th latency percentiles are reduced by more than 30%. This reduction in the tail latency is due to the DC's improved utilization factor, which leads to a much smaller number of outstanding requests, which makes it easier to absorb traffic bursts.

7 Related Work

The extensive related work in caching can be divided into two major lines of work: research caching systems (Section 7.1), and cache tuning methods (Section 7.2).

7.1 Research caching systems

Table 2 surveys 33 caching systems proposed in the research literature between 1993 and 2016. We classify these systems in terms of the per-request time complexity, the eviction and admission policies used, the support for a concurrent implementation, and the evaluation method.

Not all of the 33 caching systems fulfill the low overhead design goal of Section 2. Specifically, the complexity column in Table 2 shows that some proposals before 2002 have a computational overhead that scales logarithmically in the number of objects in the cache, which is impractical. AdaptSize differs from these systems because it has a constant complexity, and a low synchronization overhead, which we demonstrated by incorporating AdaptSize into the Varnish production system.

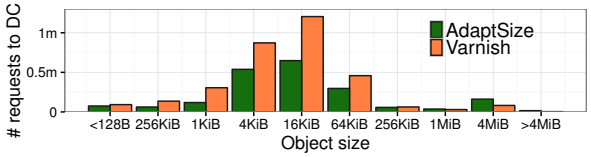


Figure 16: Comparison of the distribution of request sizes to the disk cache under a HOC running AdaptSize versus unmodified Varnish. All object sizes below 256 KiB are significantly less frequent under AdaptSize, whereas larger objects are slightly more frequent.

Of those caching systems that have a low overhead, almost none (except LRU-S and Threshold) incorporate object sizes. In particular, these systems admit and evict objects based only on recency, frequency, or a combination thereof. AdaptSize differs from these systems because it is size aware, which improves the OHR by 33-46% (as shown in Section 6.2).

There are only three low-overhead caching systems that are size aware. Threshold [2] uses a static size threshold, which has to be determined in advance. The corresponding Static policy in Section 6.3 performs poorly in our experiments. LRU-S [72] uses size-aware admission, where it admits objects with probability $1/size$. Unfortunately, this static probability is too low[6]. AdaptSize achieves a 61-78% OHR improvement over LRU-S (Figures 5 and 13). The third system [56] also uses a static parameter, and was developed in parallel to AdaptSize. AdaptSize differs from these caching systems by automatically adapting the size-aware admission parameter over time.

While most of these caching systems share our goal of improving the OHR, an orthogonal line of research seeks to achieve superior throughput using concurrent cache implementations (compare the concurrent implementation column in Table 2). AdaptSize also uses a concurrent implementation and achieves throughput comparable to production systems (Section 6.1). AdaptSize differs from these systems by improving the OHR – without sacrificing cache throughput.

The last column in Table 2 shows that most recent caching systems are evaluated using prototype implementations. Likewise, this work evaluates an actual implementation of AdaptSize (Sections 5 and 6) through experiments in a dedicated data center. We additionally use trace-driven simulations to compare to some of those systems that have only been used in simulations.

[6] We also tested several variants of LRU-S. We were either confronted with a cache tuning problem with no obvious solution (Section 3.2), or (by removing the admission component) with an OHR similar to LRU.

Name	Year	Complexity	Admission Policy	Eviction Policy	Concurrent Imp.	Evaluation
AdaptSize	**2016**	$O(1)$	**size**	**recency**	**yes**	**implementation**
Cliffhanger [14]	2016	$O(1)$	none	recency	no	implementation
Billion [47]	2015	$O(1)$	none	recency	yes	implementation
BloomFilter [49]	2015	$O(1)$	frequency	recency	no	implementation
SLRU [29]	2015	$O(1)$	none	recency+frequency	no	analysis
Lama [34]	2015	$O(1)$	none	recency	no	implementation
DynaCache [13]	2015	$O(1)$	none	recency	no	implementation
MICA [48]	2014	$O(1)$	none	recency	yes	implementation
TLRU [20]	2014	$O(1)$	frequency	recency	no	simulation
MemC3 [23]	2013	$O(1)$	none	recency	yes	implementation
S4LRU [35]	2013	$O(1)$	none	recency+frequency	no	simulation
CFLRU [62]	2006	$O(1)$	none	recency+cost	no	simulation
Clock-Pro [38]	2005	$O(1)$	none	recency+frequency	yes	simulation
CAR [7]	2004	$O(1)$	none	recency+frequency	yes	simulation
ARC [53]	2003	$O(1)$	none	recency+frequency	no	simulation
LIRS [39]	2002	$O(1)$	none	recency+frequency	no	simulation
LUV [6]	2002	$O(\log n)$	none	recency+size	no	simulation
MQ [81]	2001	$O(1)$	none	recency+frequency	no	simulation
PGDS [12]	2001	$O(\log n)$	none	recency+frequency+size	no	simulation
GD* [40]	2001	$O(\log n)$	none	recency+frequency+size	no	simulation
LRU-S [72]	2001	$O(1)$	size	recency+size	no	simulation
LRV [66]	2000	$O(\log n)$	none	frequency+recency+size	no	simulation
LFU-DA [5, 70]	2000	$O(1)$	none	frequency	no	simulation
LRFU [46]	1999	$O(\log n)$	none	recency+frequency	no	simulation
PSS [3]	1999	$O(\log n)$	frequency	frequency+size	no	simulation
GDS [11]	1997	$O(\log n)$	none	recency+size	no	simulation
Hybrid [79]	1997	$O(\log n)$	none	recency+frequency+size	no	simulation
SIZE [1]	1996	$O(\log n)$	none	size	no	simulation
Hyper [1]	1996	$O(\log n)$	none	frequency+recency	no	simulation
Log2(SIZE) [2]	1995	$O(\log n)$	none	recency+size	no	simulation
LRU-MIN [2]	1995	$O(n)$	none	recency+size	no	simulation
Threshold [2]	1995	$O(1)$	size	recency	no	simulation
2Q [41]	1994	$O(1)$	frequency	recency+frequency	no	simulation
LRU-K [58]	1993	$O(\log n)$	none	recency+frequency	no	implementation

Table 2: Historical overview of web caching systems.

7.2 Cache tuning methods

While tuning for size-based admission is entirely new, tuning has been used in other caching contexts such as tuning for the optimal balance between recency and frequency [41, 53, 46, 39, 81, 7, 9] and for the allocation of capacity to cache partitions [14, 34, 13, 69].

In these other contexts, the most common tuning approach is hill climbing with shadow caches [41, 53, 46, 39, 81, 7, 14]. Section 3.2 discusses why this approach often performs poorly when tuning size-aware admission, and Section 6 provides corresponding experimental evidence.

Another method involves a prediction model together with a global search algorithm. The most widely used prediction model is the calculation of stack distances [51, 4, 78, 77], which has been recently used as an alternative to shadow caches [69, 13, 69]. Unfortunately, the stack distance model is not suited to optimizing the parameters of an admission policy, since each admission parameter leads to a different request sequence and thus a different stack distance distribution that needs to be recalculated.

Many cache models have been studied in the CS theory community [44, 30, 15, 52, 10, 33, 17, 75, 25, 24, 36, 19, 68, 16, 37, 61, 65, 28, 80, 59, 27, 50, 8, 29, 9]. Unfortunately, all these models assume unit-sized objects. AdaptSize's Markov model allows to model size-aware admission and variable-sized objects.

8 Conclusion

AdaptSize is a new caching system for the hot object cache in CDN servers. The power of AdaptSize stems from a size-aware admission policy that is continuously optimized using a new Markov model of the HOC. In experiments with Akamai production traces, we show that AdaptSize vastly improves the OHR over both state-of-the-art production systems and research systems. We also show that our implementation of AdaptSize is robust and scalable, and improves the DC's disk utilization.

As more diverse applications with richer content migrate onto the Internet, future CDNs will experience even greater variability in request patterns and object sizes. We believe that AdaptSize and its underlying mathematical model will be valuable in addressing this challenge.

9 Acknowledgments

We thank Jens Schmitt, Sebastian Michel, our shepherd Mahesh Balakrishnan, and our anonymous reviewers for their valuable feedback.

This research is supported in part by a Google Faculty Award, and NSF grants CNS-1413998, CMMI-1538204, CMMI-1334194, and XPS-1629444.

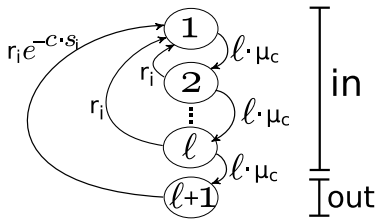
Appendix

A Proof of Theorem 1

The result in Theorem 1 is achieved by solving the Markov chain shown in Figure 8.

The key challenge when solving this chain is that the length of the LRU list changes over time. We solve this by using a mathematical convergence result.

We consider a fixed object i, and a fixed size-aware admission parameter c. Let ℓ denote the length of the LRU list. Now the Markov chain has $\ell+1$ states: one for each position in the list and one to represent the object is out of the cache, as shown below:



Over time, ℓ changes as either larger or small objects populate the cache. However, what remains constant is the expected time for an object to get evicted (if it is not requested again) as this time only depends on the overall admission rate (i.e. the size-aware admission parameter c), which is independent of ℓ. Using this insight, we modify the Markov chain to increase the push-down rate μ by a factor of ℓ: now, the expected time to traverse from position 1 to $\ell+1$ (without new requests) is constant at $1/\mu$.

We now solve the Markov chain for a fixed ℓ and obtain the limiting probability π_i of each position $i \in \{0,\ldots,\ell,\ell+1\}$. Using the π_i, we can now derive the limiting probability of being "in" the cache, $\pi_{in} = \sum_{i=0}^{\ell} \pi_i$, which can be algebraically simplified to:

$$\pi_{in} = 1 - \frac{\left(\frac{\ell}{\ell+r_i/\mu}\right)^{\ell}}{e^{-s_i/c} + \left(\frac{\ell}{\ell+r_i/\mu}\right)^{\ell} - e^{-s_i/c}\left(\frac{\ell}{\ell+r_i/\mu}\right)^{\ell}}$$

We observe that the π_{in} quickly converges in ℓ; numerically, convergence happens around $\ell > 100$. In our simulations, the cache typically holds many more objects than 100, simultaneously. Therefore, it is reasonable to always use the converged result $\ell \to \infty$. We formally solve this limit for π_{in} and obtain the closed-form solution of the long-term probability that object i is present in the cache, as stated in Theorem 1.

We remark that our convergence result uses a similar intuition as recent studies on equal-sized objects [60, 27], which is that the time it takes an object to get from position 1 to $\ell+1$ (if there are no further requests to it) converges to a constant in a LRU cache. What makes AdaptSize's model different from these models is that we consider size-aware admission and variable object sizes.

References

[1] ABRAMS, M., STANDRIDGE, C. R., ABDULLA, G., FOX, E. A., AND WILLIAMS, S. Removal policies in network caches for World-Wide Web documents. In *ACM SIGCOMM* (1996), pp. 293–305.

[2] ABRAMS, M., STANDRIDGE, C. R., ABDULLA, G., WILLIAMS, S., AND FOX, E. A. Caching Proxies: Limitations and Potentials. Tech. rep., Virginia Polytechnic Institute & State University Blacksburgh, VA, 1995.

[3] AGGARWAL, C., WOLF, J. L., AND YU, P. S. Caching on the world wide web. *IEEE Transactions on Knowledge and Data Engineering 11*, 1 (1999), 94–107.

[4] ALMASI, G., CAŞCAVAL, C., AND PADUA, D. A. Calculating stack distances efficiently. In *ACM SIGPLAN Notices* (2002), vol. 38, pp. 37–43.

[5] ARLITT, M., CHERKASOVA, L., DILLEY, J., FRIEDRICH, R., AND JIN, T. Evaluating content management techniques for web proxy caches. *Performance Evaluation Review 27*, 4 (2000), 3–11.

[6] BAHN, H., KOH, K., NOH, S. H., AND LYUL, S. Efficient replacement of nonuniform objects in web caches. *IEEE Computer 35*, 6 (2002), 65–73.

[7] BANSAL, S., AND MODHA, D. S. CAR: Clock with adaptive replacement. In *USENIX FAST* (2004), vol. 4, pp. 187–200.

[8] BERGER, D. S., GLAND, P., SINGLA, S., AND CIUCU, F. Exact analysis of TTL cache networks. *Perform. Eval. 79* (2014), 2 – 23. Special Issue: Performance 2014.

[9] BERGER, D. S., HENNINGSEN, S., CIUCU, F., AND SCHMITT, J. B. Maximizing cache hit ratios by variance reduction. *ACM SIGMETRICS Performance Evaluation Review 43*, 2 (2015), 57–59.

[10] BURVILLE, P., AND KINGMAN, J. On a model for storage and search. *Journal of Applied Probability* (1973), 697–701.

[11] CAO, P., AND IRANI, S. Cost-aware WWW proxy caching algorithms. In *USENIX symposium on Internet technologies and systems* (1997), vol. 12, pp. 193–206.

[12] CHERKASOVA, L., AND CIARDO, G. Role of aging, frequency, and size in web cache replacement policies. In *High-Performance Computing and Networking* (2001), pp. 114–123.

[13] CIDON, A., EISENMAN, A., ALIZADEH, M., AND KATTI, S. Dynacache: dynamic cloud caching. In *USENIX HotCloud* (2015).

[14] CIDON, A., EISENMAN, A., ALIZADEH, M., AND KATTI, S. Cliffhanger: Scaling performance cliffs in web memory caches. In *USENIX NSDI* (2016), pp. 379–392.

[15] COFFMAN, E. G., AND DENNING, P. J. *Operating systems theory*. Prentice-Hall, 1973.

[16] COFFMAN, E. G., AND JELENKOVIĆ, P. Performance of the move-to-front algorithm with Markov-modulated request sequences. *Operations Research Letters 25* (1999), 109–118.

[17] DAN, A., AND TOWSLEY, D. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *ACM SIGMETRICS* (1990), pp. 143–152.

[18] DILLEY, J., MAGGS, B. M., PARIKH, J., PROKOP, H., SITARAMAN, R. K., AND WEIHL, W. E. Globally distributed content delivery. *IEEE Internet Computing 6*, 5 (2002), 50–58.

[19] DOBROW, R. P., AND FILL, J. A. The move-to-front rule for self-organizing lists with Markov dependent requests. In *Discrete Probability and Algorithms*. Springer, 1995, pp. 57–80.

[20] EINZIGER, G., AND FRIEDMAN, R. Tinylfu: A highly efficient cache admission policy. In *IEE Euromicro PDP* (2014), pp. 146–153.

[21] ELAARAG, H., ROMANO, S., AND COBB, J. *Web Proxy Cache Replacement Strategies: Simulation, Implementation, and Performance Evaluation*. Springer Briefs in Computer Science. Springer London, 2013.

[22] ERA—TRENDS, C. V. G. I. T. F. T. Z., AND ANALYSIS. CISCO VNI global IP traffic forecast: The zettabyte era—trends and analysis, May 2015. Available at `http://goo.gl/wxuvVk`, accessed 09/12/16.

[23] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *USENIX NSDI* (2013), pp. 371–384.

[24] FILL, J. A., AND HOLST, L. On the distribution of search cost for the move-to-front rule. *Random Structures & Algorithms 8* (1996), 179–186.

[25] FLAJOLET, P., GARDY, D., AND THIMONIER, L. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics 39* (1992), 207–229.

[26] FOFACK, N. C., DEHGHAN, M., TOWSLEY, D., BADOV, M., AND GOECKEL, D. L. On the performance of general cache networks. In *VALUETOOLS* (2014), pp. 106–113.

[27] FRICKER, C., ROBERT, P., AND ROBERTS, J. A versatile and accurate approximation for LRU cache performance. In *ITC* (2012), p. 8.

[28] GALLO, M., KAUFFMANN, B., MUSCARIELLO, L., SIMONIAN, A., AND TANGUY, C. Performance evaluation of the random replacement policy for networks of caches. In *ACM SIGMETRICS/ PERFORMANCE* (2012), pp. 395–396.

[29] GAST, N., AND VAN HOUDT, B. Transient and steady-state regime of a family of list-based cache replacement algorithms. In *ACM SIGMETRICS* (2015), pp. 123–136.

[30] GELENBE, E. A unified approach to the evaluation of a class of replacement algorithms. *IEEE Transactions on Computers 100* (1973), 611–618.

[31] GODARD, S. Iostat, 2015. Available at `http://goo.gl/JZmbUp`, accessed 09/12/16.

[32] GRAZIANO, P. Speed up your web site with Varnish. *Linux Journal 2013*, 227 (2013), 4.

[33] HENDRICKS, W. The stationary distribution of an interesting Markov chain. *Journal of Applied Probability* (1972), 231–233.

[34] HU, X., WANG, X., LI, Y., ZHOU, L., LUO, Y., DING, C., JIANG, S., AND WANG, Z. LAMA: Optimized locality-aware memory allocation for key-value cache. In *USENIX ATC* (2015), pp. 57–69.

[35] HUANG, Q., BIRMAN, K., VAN RENESSE, R., LLOYD, W., KUMAR, S., AND LI, H. C. An analysis of Facebook photo caching. In *ACM SOSP* (2013), pp. 167–181.

[36] JELENKOVIĆ, P. R. Asymptotic approximation of the move-to-front search cost distribution and least-recently used caching fault probabilities. *The Annals of Applied Probability 9* (1999), 430–464.

[37] JELENKOVIĆ, P. R., AND RADOVANOVIĆ, A. Least-recently-used caching with dependent requests. *Theoretical computer science 326* (2004), 293–327.

[38] JIANG, S., CHEN, F., AND ZHANG, X. CLOCK-Pro: An effective improvement of the clock replacement. In *USENIX ATC* (2005), pp. 323–336.

[39] JIANG, S., AND ZHANG, X. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS 30*, 1 (2002), 31–42.

[40] JIN, S., AND BESTAVROS, A. GreedyDual* web caching algorithm: exploiting the two sources of temporal locality in web request streams. *Computer Communications 24* (2001), 174–183.

[41] JOHNSON, T., AND SHASHA, D. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB* (1994), pp. 439–450.

[42] KAMP, P.-H. Varnish notes from the architect, 2006. Available at `https://www.varnish-cache.org/docs/trunk/phk/notes.html`, accessed 09/12/16.

[43] KAMP, P.-H. Varnish LRU architecture, June 2007. Available at `https://www.varnish-cache.org/trac/wiki/ArchitectureLRU`, accessed 09/12/16.

[44] KING, W. F. Analysis of demand paging algorithms. In *IFIP Congress (1)* (1971), pp. 485–490.

[45] LE BOUDEC, J.-Y., MCDONALD, D., AND MUNDINGER, J. A generic mean field convergence result for systems of interacting objects. In *Quantitative Evaluation of Systems* (2007), IEEE, pp. 3–18.

[46] LEE, D., CHOI, J., KIM, J.-H., NOH, S. H., MIN, S. L., CHO, Y., AND KIM, C. S. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *ACM SIGMETRICS* (1999), vol. 27, pp. 134–143.

[47] LI, S., LIM, H., LEE, V. W., AHN, J. H., KALIA, A., KAMINSKY, M., ANDERSEN, D. G., SEONGIL, O., LEE, S., AND DUBEY, P. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *ACM ISCA* (2015), pp. 476–488.

[48] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. MICA: A holistic approach to fast in-memory key-value storage. In *USENIX NSDI* (2014), pp. 429–444.

[49] MAGGS, B. M., AND SITARAMAN, R. K. Algorithmic nuggets in content delivery. *ACM SIGCOMM CCR 45* (2015), 52–66.

[50] MARTINA, V., GARETTO, M., AND LEONARDI, E. A unified approach to the performance analysis of caching systems. In *IEEE INFOCOM* (2014).

[51] MATTSON, R. L., GECSEI, J., SLUTZ, D. R., AND TRAIGER, I. L. Evaluation techniques for storage hierarchies. *IBM Systems journal 9*, 2 (1970), 78–117.

[52] MCCABE, J. On serial files with relocatable records. *Operations Research 13* (1965), 609–618.

[53] MEGIDDO, N., AND MODHA, D. S. ARC: A self-tuning, low overhead replacement cache. In *USENIX FAST* (2003), vol. 3, pp. 115–130.

[54] Modern network design, November 2016. Available at `https://www.fastly.com/products/modern-network-design`, accessed 02/17/17.

[55] MOTWANI, R., AND RAGHAVAN, P. *Randomized algorithms*. Chapman & Hall/CRC, 2010.

[56] NEGLIA, G., CARRA, D., FENG, M., JANARDHAN, V., MICHIARDI, P., AND TSIGKARI, D. Access-time aware cache algorithms. In *IEEE ITC* (2016), vol. 1, pp. 148–156.

[57] NYGREN, E., SITARAMAN, R. K., AND SUN, J. The Akamai Network: A platform for high-performance Internet applications. *ACM SIGOPS Operating Systems Review 44*, 3 (2010), 2–19.

[58] O'NEIL, E. J., O'NEIL, P. E., AND WEIKUM, G. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD 22*, 2 (1993), 297–306.

[59] O'NEIL, E. J., O'NEIL, P. E., AND WEIKUM, G. An optimality proof of the LRU-K page replacement algorithm. *JACM 46* (1999), 92–112.

[60] OSOGAMI, T. A fluid limit for a cache algorithm with general request processes. *Advances in Applied Probability 42*, 3 (2010), 816–833.

[61] PANAGAKIS, A., VAIOS, A., AND STAVRAKAKIS, I. Approximate analysis of LRU in the case of short term correlations. *Computer Networks 52* (2008), 1142–1152.

[62] PARK, S.-Y., JUNG, D., KANG, J.-U., KIM, J.-S., AND LEE, J. CFLRU: a replacement algorithm for flash memory. In *ACM/IEEE CASES* (2006), pp. 234–241.

[63] PETRUSHEV, P. P., AND POPOV, V. A. *Rational approximation of real functions*, vol. 28. Cambridge University Press, 2011.

[64] POŠÍK, P., HUYER, W., AND PÁL, L. A comparison of global search algorithms for continuous black box optimization. *Evolutionary computation 20*, 4 (2012), 509–541.

[65] PSOUNIS, K., ZHU, A., PRABHAKAR, B., AND MOTWANI, R. Modeling correlations in web traces and implications for designing replacement policies. *Computer Networks 45* (2004), 379–398.

[66] RIZZO, L., AND VICISANO, L. Replacement policies for a proxy cache. *IEEE/ACM TON 8* (2000), 158–170.

[67] ROCCA, E. Running Wikipedia.org, June 2016. Available at `https://www.mediawiki.org/wiki/File:WMF_Traffic_Varnishcon_2016.pdf`, accessed 09/12/16.

[68] RODRIGUES, E. R. The performance of the move-to-front scheme under some particular forms of Markov requests. *Journal of applied probability* (1995), 1089–1102.

[69] SAEMUNDSSON, T., BJORNSSON, H., CHOCKLER, G., AND VIGFUSSON, Y. Dynamic performance profiling of cloud caches. In *ACM SoCC* (2014), pp. 1–14.

[70] SHAH, K., MITRA, A., AND MATANI, D. An O(1) algorithm for implementing the LFU cache eviction scheme. Tech. rep., Stony Brook University, 2010.

[71] SITARAMAN, R. K., KASBEKAR, M., LICHTENSTEIN, W., AND JAIN, M. Overlay networks: An Akamai perspective. In *Advanced Content Delivery, Streaming, and Cloud Services*. John Wiley & Sons, 2014.

[72] STAROBINSKI, D., AND TSE, D. Probabilistic methods for web caching. *Perform. Eval. 46* (2001), 125–137.

[73] TANGE, O. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine 36*, 1 (Feb 2011), 42–47.

[74] Alexa top sites on the web, March 2016. `http://www.alexa.com/topsites`, accessed 03/16/16.

[75] TSUKADA, N., HIRADE, R., AND MIYOSHI, N. Fluid limit analysis of FIFO and RR caching for independent reference model. *Perform. Eval. 69* (Sept. 2012), 403–412.

[76] VELÁZQUEZ, F., LYNGSTØL, K., FOG HEEN, T., AND RENARD, J. *The Varnish Book for Varnish 4.0*. Varnish Software AS, March 2016.

[77] WALDSPURGER, C. A., PARK, N., GARTHWAITE, A., AND AHMAD, I. Efficient MRC construction with SHARDS. In *USENIX FAST* (2015), pp. 95–110.

[78] WIRES, J., INGRAM, S., DRUDI, Z., HARVEY, N. J., AND WARFIELD, A. Characterizing storage workloads with counter stacks. In *USENIX OSDI* (2014), pp. 335–349.

[79] WOOSTER, R. P., AND ABRAMS, M. Proxy caching that estimates page load delays. *Computer Networks and ISDN Systems 29*, 8 (1997), 977–986.

[80] YOUNG, N. E. Online paging against adversarially biased random inputs. *Journal of Algorithms 37* (2000), 218–235.

[81] ZHOU, Y., PHILBIN, J., AND LI, K. The multi-queue replacement algorithm for second level buffer caches. In *USENIX ATC* (2001), pp. 91–104.

Bringing IoT to Sports Analytics

Mahanth Gowda[1], Ashutosh Dhekne[1,†], Sheng Shen[1,†], Romit Roy Choudhury[1],
Xue Yang[2], Lei Yang[2], Suresh Golwalkar[2], and Alexander Essanian[2]

[1]University of Illinois at Urbana-Champaign
[2]Intel
[†]*Co-secondary authors*

Abstract

This paper explores the possibility of bringing IoT to sports analytics, particularly to the game of Cricket. We develop solutions to track a ball's 3D trajectory and spin with inexpensive sensors and radios embedded in the ball. Unique challenges arise rendering existing localization and motion tracking solutions inadequate. Our system, *iBall*, mitigates these problems by fusing disparate sources of partial information – wireless, inertial sensing, and motion models – into a non-linear error minimization framework. Measured against a *mm*-level ground truth, the median ball location error is at 8*cm* while rotational error remains below 12° even at the end of the flight. The results do not rely on any calibration or training, hence we expect the core techniques to extend to other sports like baseball, with some domain-specific modifications.

1 Introduction

Sports analytics is a thriving industry in which motion patterns of balls, racquets, and players are being analyzed for coaching, strategic insights, and predictions. The data for such analytics are sourced from expensive high-quality cameras installed in stadiums, processed at powerful backend servers and clouds. We explore the possibility of significantly lowering this cost barrier by embedding cheap Inertial Measurement Unit (IMU) sensors and ultrawide band (UWB) radios inside balls and players' shoes. If successful, real-time analytics should be possible anytime, anywhere. Aspiring players in local clubs could read out their own performance from their smartphone screens; school coaches could offer quantifiable feedback to their students.

Our work follows a growing excitement in IoT based sports analytics. Sensor-enabled football helmets, aimed at detecting concussions and head injuries, are already in the market. Nike is prototyping IMU-embedded shoes [34, 47], while multiple startups are pursuing ideas around camera-embedded jerseys [5], GPS-enabled soccer balls [6], and bluetooth frisbees [1]. However, we have not found a serious effort to accurately characterize 3D ball motion, such as trajectory, orientation, revolutions per second, etc.

The rich literature in wireless localization and inertial gesture recognition does not apply directly. WiFi-like localization infrastructure is mostly missing in the playground, and even if deployed, is not designed to support *cm-scale* 3D location at ball speeds. Inertial sensors such as accelerometers do not measure gravity when the ball is in free fall, since these sensors detect only reactive forces. Worse, gyroscopes saturate at around 6 revolutions per second (rps) [8], while even an amateur player can spin the ball at 12*rps*. In general, tracking a fast moving/spinning object in an open playground presents a relatively unexplored context, distinct from human-centric localization and gesture tracking applications.

In approaching this problem top-down, we develop multiple wireless and sensing modules, and engineer them into a unified solution. The technical core of our system relies on using ultrawide band (UWB) radios to compute the time of flight (ToF) and angle of arrival (AoA) of the signals from the ball. When this proves inadequate, we model the ball's physical motion as additional constraints to the underdetermined system of equations. Finally, we fuse all these sources of information into a non-linear error minimization framework and extract out the parameters of ball trajectory.

Spin estimation poses a different set of challenges. We need to determine the initial orientation of the ball at its release position and then track the 3D rotation through the rest of the flight. With unhelpful accelerometers and gyroscopes, we are left with magnetometers. While magnetometers do not capture all the dimensions of rotation, we recognize that the uncertainty in the ball's spin is somewhat limited since air-drag is the only source of torque. This manifests on the magnetometer as a sinusoidal signal, with a time varying bias (called "wobble"). We formulate this as a curve-fitting problem, and jointly resolve the ball's angular velocity as well as "wobble". In general, we learn that magnetometers can serve as gyroscopes in free-spinning objects.

Our experiment platform is composed of an Intel Curie board (IMU + UWB) embedded in the ball by a professional design company [3]. Two small UWB receiver boxes, called *anchors*, are also placed on the ground – additional anchors are infeasible due to the field layout

in the Cricket game, discussed shortly. For ground truth, we use 8 Vicon based IR cameras positioned at 4 corners of the ceiling. IR markers are pasted on the ball to enable precise tracking ($0.1mm$ and 0.2° for location and orientation). Since the ViCon coverage area is $10x10x4m^3$ – around half of the actual Cricket trajectories – we scale-down the length of the throws while maintaining realistic speed and spin.

Reported results from 100 different throws achieve median location accuracy of $8cm$ and orientation errors of 11.2°, respectively. A player (wearing a clip-on UWB board) is also tracked with a median error of $1.2m$ even when he is at the periphery of the field ($80m$ away from the anchor). All results are produced at sub-second latency, adequate for real time feedback to human players.

There is obviously room for continued research and improvement. First, we have sidestepped the energy question. In future, perhaps wireless charging will mitigate this problem; perhaps fast rotation will automatically scavenge energy. For now, our solution allows a battery life of ≈ 75 minutes between re-charges, permitting short training sessions. Second, our aerodynamic motion models are simplistic and did not get stress-tested in indoor settings– this may have yielded favorable results. Moreover, we could not exceed throw speeds beyond 45 miles/hour and 12 revolutions/s, both of which are around half of the professionals. Finally, this paper focuses on Cricket, and although we believe our techniques are generalizable with modest modifications, we have not verified these claims. Our ongoing work is focussed on adapting *iBall* to baseball and frisbee.

To summarize, the contributions of this paper are:

- *Formulating object tracking as an information fusion problem under the practical constraints of Cricket.* Designing an optimization framework for fusing time of flight (ToF) measurements, air-drag motion models, and noisy angle of arrival (AoA) estimates, to ultimately achieve desired accuracy.
- *Identifying various opportunities arising from free-fall motion.* Harnessing the magnetometer to jointly estimate rotation and rotation axis, thereby emulating an inertial gyroscope in free-fall scenarios.

The rest of the paper expands on these technical components woven together by significant engineering effort. We begin with some background on Cricket, followed by challenges, opportunities, design, and implementation.

2 Background and Platform

2.1 Quick Primer on Cricket

We summarize the basic rules of cricket for those unfamiliar with the game. A Cricket match is divided into two sessions – in any session, one team is called the *batting side* and the other is called the *bowling* or *fielding side*. The teams switch roles in the second session. A playing *pitch* is located at the center of the field, with two *wickets* on each side of the pitch. A wicket is a set of 3 wooden sticks placed vertically one beside the other (see Fig.1). A player from the batting side stands in front of a *wicket* while a player from the bowling side runs up to the other wicket and throws the ball towards the batsman. All other players of the bowling side are called *fielders* and stand scattered around the park.

The bowler's objective is to hit the wicket with the ball, or to force the batsman to hit the ball in a way that a fielder can *catch* the ball before it drops to the ground. If the bowler is successful, the batsman is *out*, i.e., he goes off the field and the next batsman of the batting team comes to face the bowler. The batsman's goal, on the other hand, is to not get out, and to also hit the ball so that it goes past the fielders and reaches the periphery of the park, called a *boundary*. If the ball bounces on the ground at least once before it crosses the boundary, then the batting side scores 4 more points (called *runs*); if the ball goes over the boundary without any bounce, 6 runs are added to the team's score. A session ends when either N deliveries have been bowled or all the 11 batsmen are out, whichever occurs earlier. At the end , the team with a higher total score wins.

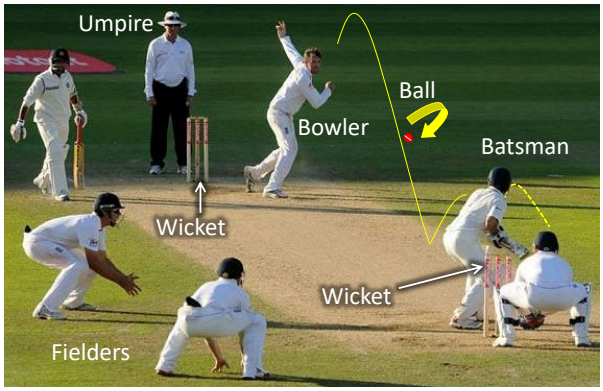


Figure 1: Cricket in action. Two sets of wickets placed at the bowler's and batsman's end.

Analogy to Baseball: The similarity between Cricket and Baseball, from the perspective of ball and player tracking, is noteworthy. The baseball travels and spins at comparable speeds and rates, while the length of the "pitch" is also similar. Differences might arise from the stitching patterns on the ball, and the viability of placing multiple anchors in baseball (in contrast to 2 in Cricket). In this sense, perhaps the ball's trajectory tracking problem becomes simpler in the case of baseball, but the spinning question still remains relevant.

Figure 2: Ball instrumentation: (a) Intel Curie board with IMU sensors and UWB radio. (b) Scooped out cricket ball for snug fit of the sensor box. (c) Closed ball with sensor box. (d) UWB 4-antenna MIMO radio serving as an anchor.

2.2 The Solution Space

There are obviously many approaches to ball and player tracking – we briefly discuss our deliberations for selecting the IMU/UWB platform.

- High end camera networks used today are expensive ($\$100,000+$) because they need to be far away [9], hence, we considered placing cheaper cameras at the wickets. The benefit is that the ball need not be instrumented. However, with no markers on the ball, spin tracking and de-blurring is challenging even with the best cameras. Cheap cameras at the wickets experience similar problems, get occasionally occluded by players, suffer in low light, and cannot track fielders scattered in the field. Experiments with iPhone cameras yielded poor results even with colored tapes on the ball.

- RFIDs on the ball (and readers placed at wickets) pose a far less price point ($2000). However, the rapidly spinning RFIDs exhibit continuous disconnections [44]. Further, cricket balls are continuously rubbed to maintain shine, crucial to the ball's swing and spin – pasting antennas on the surface is impractical.

- WiFi based tracking solutions are also impractical under the constraints of high speed and spin, cm-scale accuracy, and availability of a very few base stations on the 2D ground (which makes 3D tracking difficult due to *dilution of precision* (DoP) [26, 39]). Trials with laser rangers [22] and acoustic reflection techniques [20] also proved pointless. Given the small cross-sectional area of the ball, the reflections from them yielded high false positives.

- Our choice to embed electronics in the ball, although cumbersome, proved practical for accuracy and coverage in the field. In discussion with 3D printing and design companies, we gained confidence that embedding should be feasible even under impact. Finally, UWB radios offer time of flight capabilities, a pre-requisite for extremely fast moving balls ($> 80+$ miles/hour). Our overall cost is estimated at $250.

2.3 Instrumenting Balls and Anchors

Fig.2 illustrates the steps in ball instrumentation. A Quark CPU, IMU BMM150 sensors, and a Decawave UWB radio are cased in a plastic polymer box and snug-fitted into a hole (to avoid rattling). The two halves of the ball are closed shut and a hole drilled to bring out a USB port to the surface for recharging. The sensor data is stored on a local flash or can be streamed through the UWB radio to the nearby "anchor".

The anchor is a UWB receiver box placed at each wicket. The UWB radio from Decawave [4] is 802.15.4 compliant with support for 3.5 to 6.5 GHz bands (12 channels) and a bandwidth of 500 MHz (data rates of up to 6.8 Mbps). The radio operates under low power, with sleep current at 100 nanoAmp. While the ball contains a single antenna (due to space restrictions), a 4 antenna MIMO radio is fitted in the anchor (Fig.2(c)).

Fig.3 illustrates the overall deployment in real settings. UWB signals are exchanged between the ball and anchors to compute the ball's range as well as the angle of arrival (AoA) from the phase differences at different antennas. The range and AoA information are combined for trajectory estimation. For spin analytics, the sensors inside the ball send out the data for off-ball processing. Players in the field can optionally wear the same IMU/UWB device (as in the ball) for 2D localization and tracking.

3 System Design: 3D Spin Tracking

Three main spin-related metrics are of interest to Cricketers: (1) revolutions per second, (2) rotation axis, and (3) seam plane[1]. From a sensing perspective, all these 3 metrics can be derived if the ball's 3D orientation can be tracked over time. This motivates a discussion on orientation, rotation, and coordinate frameworks.

[1]The seam is a stitch along the equator of the Cricket ball.

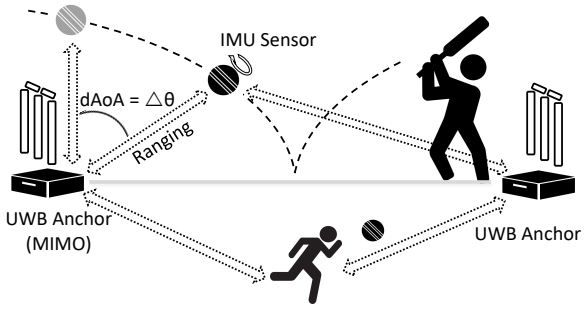


Figure 3: Two anchors and a ball deployed on the ground, while players optionally have the device in their shoes.

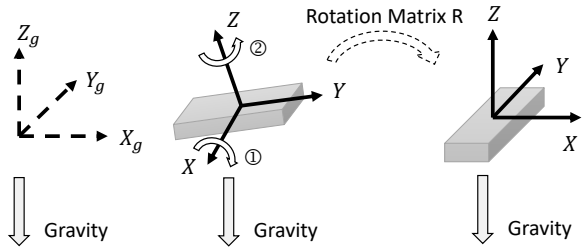


Figure 4: Rotating local axes to align local directions of gravity and magnetic North with the global directions.

3.1 Foundations of Orientation

The *orientation* of an object is the representation of the object's local X, Y, Z axes as vectors in the global coordinate frame. A *rotation* of an object is a change of orientation, and can be decomposed into sequence of rotations around its (local) X, Y, and Z axes. Put differently, any new orientation can be achieved by rotating the object (by appropriate amounts) on each of the 3 axes, one after the other. The gyroscope measures each of these rotations per unit time, called *angular velocity*. Thus, theoretically, if one knows the *initial orientation* of an object in the global coordinate frame, then subsequent orientations can be tracked by integrating the gyroscope–measured angular velocity across time.

Expressing the object's initial orientation in the global framework should be possible since gravity and magnetic North are both along globally known directions. Thus, the object's local axes can be rotated until the local representation of gravity and North align with the known global directions. We consider an example below.

Fig.4(a) shows a global frame $\{X_g, Y_g, Z_g\}$ with its X_g pointing East, Y_g pointing North, and Z_g pointing up against gravity. Fig.4(b) shows an object in an unknown orientation. Now, the object can be rotated around X axis until the measured gravity is along its own $-Z$ direction; it can be rotated again around this $-Z$ axis until the measured magnetic field (compass) is along its own Y. Now the local and global frameworks have fully aligned, and we denote the total rotation as a single matrix R:

$$\begin{bmatrix} X & Y & Z \end{bmatrix} R = \begin{bmatrix} X_g & Y_g & Z_g \end{bmatrix}$$

We define an object's orientation, R_O, as the inverse of this rotation matrix, R^{-1}. Intuitively, if an object needs a clockwise rotation of 30° to align with the global framework, then its orientation must be 30° counter-clockwise. Thus, we have the capability to compute both initial orientation and angular velocity; from these, any spin-related analytics should ideally be trackable.

■ **Challenges with In-Flight Balls:** Challenges emerge in the real world and particularly in this cricket setting: (1) The gyroscope is noisy and this error accumulates since rotation is a time–integral of angular velocity (2) Worse, the gyroscope saturates beyond 5 revolutions/sec. (rps), while even amateurs can spin the ball at $12rps$ (professionals attain > 30). (3) Finally, gravity is not measured in accelerometers during free-fall which precludes opportunities to rotate and align the local coordinate frame [2]. In sum, known techniques cannot compute initial orientation or rotation when the ball is in flight.

3.2 The Core Opportunity

At a high level, 2 observations are central.

- In the absence of air-flow, there is no external torque on the ball, implying that the ball's rotation is restricted to a single axis throughout the flight (i.e., the axis around which it was rotated by the bowler).
- From the ball's local reference frame, the magnetic North vector spins around some axis. Given a single rotation axis, the magnetometer can indeed infer the axis and measure both magnitude and direction of rotation.

Of course, air-drag pollutes this opportunity since the ball begins to experience additional rotations. This poses the main challenge. An illustrative example follows.

3.3 An Illustrative Example

Let's assume the ball's mass is symmetrically distributed and its center of mass is precisely at the center. Let's also consider gravity forces alone and no air drag. Now, due to conservation of angular momentum, the ball will not change its rotating state because no torque is generated from gravity. The dimension of this rotational motion is limited to 1 since the motion can be continuously expressed around a single axis, R^G. Fig.5 illustrates the situation – each local X, Y, Z axis rotates in different cones

[2] A full analysis of IMU deficiencies is outside the scope of this paper. Briefly, gyroscope saturation is rooted in imperfections accumulated during the manufacturing process of MEMS sensors. These imperfections make gyroscopes exhibit nonlinear responses to input vibrations [48, 40], and attempts to reduce imperfections increases cost or lowers yield rate. Accelerometers on the other hand are fundamentally designed to measure reactive forces, and hence, do not sense gravity during free fall.

around the same R^G. As an aside, the magnetic North is also a fixed vector N^G in the global framework (henceforth, we use superscript G/L to indicate that a vector is being observed in the global/local framework).

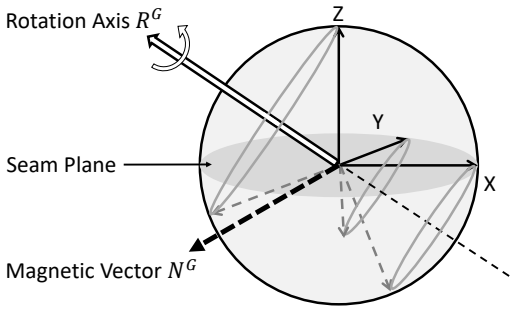


Figure 5: In the global framework, the ball in rotating around a constant rotation axis R^G.

Shifting our perspective from the global to local coordinate system, Fig.6(a) shows that the local X, Y, and Z axes are now fixed, but the magnetic vector N^L rotates in a cone around a fixed local vector R^L. Since magnetometers can reliably measure a single dimension of rotation, it should be possible to measure the parameters of this cone. This is the core opportunity – the low-dimensional mobility during free-fall empowers the magnetometer to serve as a gyroscope.

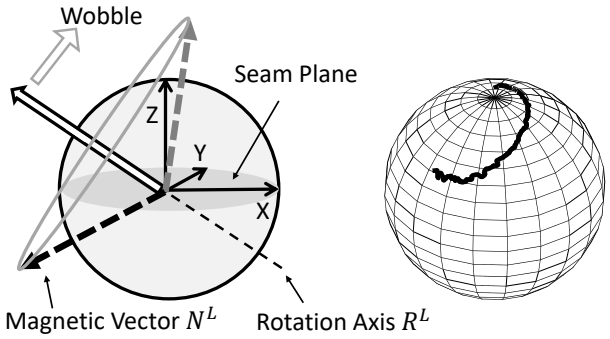


Figure 6: (a) In the local framework, the magnetic vector N^L is rotating around local rotation axis R^L. (b) In the local framework, local rotation axis R^L is slowly moving

Unfortunately, with air-drag, the ball still continues to rotate around the same global axis R^G, but experiences an additional rotation along a changing axis. To envision this, consider the ball spinning around the global vertical axis with the seam on the horizontal plane. With air-drag, the ball can continue to spin around the identical vertical axis, but the seam plane can gradually change to lie on the vertical plane. This is called "wobble" and can be modeled as a varying local rotation axis, R^L. Fig.6(b) shows the locus of R^L as it moves in the local framework (this was derived from ViCon ground truth). Thus, the center of the N^L cone is moving on the sphere surface, even though the width of the cone remains unchanged. This derails the ability to compute rotations from magnetometers.

3.4 Problem Formulation

Based on the earlier discussion, we know that if two non-collinear vectors can be observed in the local framework, and their representations known in the global frame, then the orientation of the object can be resolved. We mathematically express the orientation of the ball at time t as a rotation matrix, $R_{O(t)}$. This matrix is a function of the globally fixed vectors (i.e., rotation axis and magnetic North) and their locally measured counterparts.

$$R_{O(t)} = \begin{bmatrix} R^G & N^G & R^G \times N^G \end{bmatrix} \begin{bmatrix} R^L_{(t)} & N^L_{(t)} & R^L_{(t)} \times N^L_{(t)} \end{bmatrix}^{-1} \quad (1)$$

Here R^G and N^G are the rotation axis and magnetic North vectors, respectively – both are in the global framework and are constant during flight. The third column vector, $(R^G \times N^G)$, is a cross product necessary to equalize the matrix dimensions on both sides. $N^L_{(t)}$ is the local magnetic vector measured by the magnetometer, $\begin{bmatrix} m_x & m_y & m_z \end{bmatrix}^T$. $R^L_{(t)}$ is the local rotation axis which is slowly changing during the flight of the ball. From previous discussion we know that $R^L_{(t)}$ is always the centerline of the instantaneous $N^L_{(t)}$ cone.

Our goal now is to estimate two of the unknowns, namely R^G and time varying $R^L_{(t)}$. We know that R^G remains constant hence resolving it at the beginning of the flight will suffice – the same value can be used all the way till the end. For $R^L_{(t)}$, we know that it is moving on the sphere of the ball and the magnetic North is constantly rotating around it. We focus on tracking $R^L_{(t)}$ first and then address R^G.

3.5 Tracking Local Rotation Axis $R^L_{(t)}$

Since $N^L_{(t)}$ forms a cone around $R^L_{(t)}$, tracking $R^L_{(t)}$ is equivalent to tracking the centerline of the cone. Now, given that 3 non-coplanar unit vectors determine a cone, a straightforward idea is to fit a cone using 3 consecutive measurements: $N^L_{(t-1)}$, $N^L_{(t)}$ and $N^L_{(t+1)}$. Fig.7 shows the result: the estimation follows the true $R^L_{(t)}$ trend, but is considerably noisy.

These noise in $R^L_{(t)}$ estimation will translate to orientation error according to Equation 1. The noise cannot be reduced by fitting the cone over larger number of magnetometer measurements – this is because the cone would have moved considerably within a few sampling intervals. Our observation is that, because the ball's flight

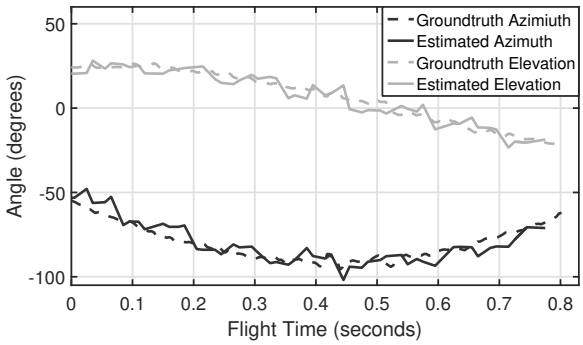


Figure 7: One example of $R^L_{(t)}$ estimation from cone fitting

time is short (less than a second), we can effectively describe the (azimuth and elevation[3]) changes in $R^L_{(t)}$ as a quadratic function of time t. Formally:

$$R^L_{(t)} = \begin{bmatrix} \cos(\theta_t)\cos(\varphi_t) \\ \cos(\theta_t)\sin(\varphi_t) \\ \sin(\theta_t) \end{bmatrix}$$

$$\text{Elevation } \theta_t = A_{el}t^2 + B_{el}t + C_{el}$$

$$\text{Azimuth } \varphi_t = A_{az}t^2 + B_{az}t + C_{az}$$

Put differently, we model the motion of a moving cone, under the constraints that the center of cone is moving on quadratic path (on the surface of the sphere) and that the cone angle $\theta_{NR} = \angle\left(N^L_{(t)}, R^L_{(t)}\right)$ is constant. We pick the best 6 parameters of this model that minimize the *variance* of the cone angles as measured over time. Our objective function is:

$$\underset{6\,\text{paras}}{\text{argmin}}\ \text{Var}\left[\angle\left(R^L_{(T)}, N^L_{(T)}\right), \angle\left(R^L_{(T+1)}, N^L_{(T+1)}\right), \cdots\right] \tag{2}$$

where T is the moment the ball is released. The initial condition to this optimization function is derived from a smoothened version of the basic cone fitting approach, described in Fig.7.

3.6 Track Global Rotation Axis R^G

Fig.8 shows 2 phases of ball tracking: *pre-flight* and *in-flight*. The above section described phase 2, the tracking of $R^L_{(t)}$ when the ball is spinning *in-flight*. However, recall that the global rotation axis, R^G, also needs to be estimated to solve for orientation $R_{O(t)}$ in Equation 1.

Tracking R^G during the ball's flight is difficult. Sensor data during the flight only tells us where $R^L_{(t)}$ is pointing (center of the $N^L_{(t)}$ cone) but it does not reveal any information about R^G. Fortunately, the rotation axis R^G and

[3]Azimuth and elevation are latitudinal and longitudinal directions on a sphere's surface: a point on the sphere can be expressed as a tuple of these 2 angles.

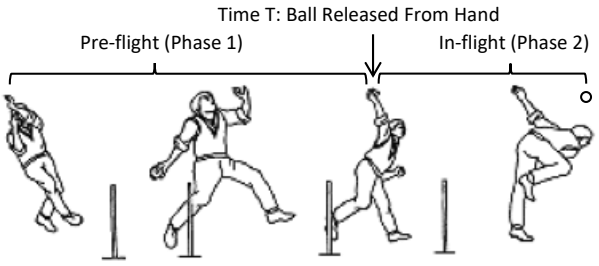


Figure 8: Two phases of ball motion.

magnetic vector N^G are two constant vectors. The angle between these two vectors, $\angle\left(N^G, R^G\right)$ is the same as the local $N^L_{(t)}$ cone angle θ_{NR}. Thus, R^G can only lie on a cone around N^G whose cone angle is θ_{NR}. Although useful, it's insufficient – we still do not know the point on the cone's circle corresponding to R^G.

To mitigate this problem, we focus on sensor measurements in phase 1 (pre-flight). Since this is not free-fall, and the ball is not spinning fast, the gyroscope and accelerometer are both useful. Our aim is to identify a stationary time point to compute the initial orientation of the ball, and use the gyroscope thereafter to integrate rotation until the point of release, T. Once we obtain orientation at T, denoted $R_{O(T)}$, we simply use the following equation to solve for the global rotation axis $R^G_{(T)}$

$$R^G_{(T)} = R_{O(T)}R^L_{(T)} \tag{3}$$

Then, we use $R^G_{(T)}$ as our estimation of R^G for the whole flight in Phase 2.

In general, gyroscope noise and saturation can render $R^G_{(T)}$ erroneous. However, since the ball does not spin while in the hand (in fact, it rotates less than 1 revolution), and the angular velocity saturates the gyroscope only at the last few moments before ball-release, we calibrate $R^G_{(T)}$ using the cone angle restriction mentioned above. Fig.9 reports consistently small $R^G_{(T)}$ error from 50 experiments.

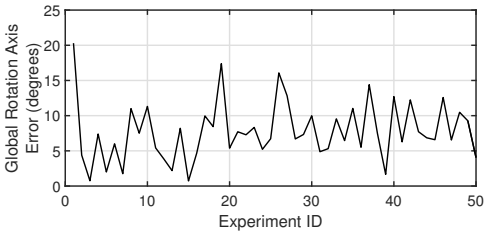


Figure 9: Error in estimating global rotation axis $R^G_{(T)}$ is reasonably small across 50 experiments.

In conclusion, gyroscope dead-reckoning right before ball release, combined with local rotation axis tracking

during flight, together yields the time-varying orientation of the ball. Algorithm 1 presents the pseudo code for final overall solution.

Algorithm 1 Ball Orientation Tracking During Flight

1: Get coarse $R^L_{(t)}$ by combining 3 consecutive magnetometer measurements
2: Use them as the initial starting point to search for parameters that minimize Var $\left[N^L_{(t)}\text{ cone angles}\right]$
3: Compute cone angle $\theta_{RN} = \text{Mean}\left[N^L_{(t)}\text{ cone angles}\right]$
4: Use gyroscope to tracking ball's orientation at the release time, $R_{O(T)}$
5: Get global rotation axis during flight: $R^G = R_{O(T)}\,R^L_{(T)}$
6: Calibrate R^G using θ_{RN}.
7: Use Equation 1 to compute ball's orientation at any time t during flight

4 System Design: 3D Trajectory Tracking

Location related analytics are also of interest in Cricket. 3 main metrics are: (1) distance to first bounce, called *length*, (2) direction of ball motion, called *line*, and (3) *speed* of the ball at the end of the flight. These metrics are all derivatives of the ball's *3D trajectory*. Our approach to estimating 3D trajectory relies on formulating a parametric model of the trajectory, as a fusion of the *time of flight* (ToF) of UWB signals, *angle of arrival* (AoA), physics motion models, and DoP constraints (explained later). A gradient decent approach minimizes a non-linear error function, resulting in an estimate of the trajectory. We present technical details next.

4.1 Ranging with UWB

The Decawave UWB radios offer time resolution at 15.65ns. With modest engineering, we were able to compute the ToF and translate it to range measurements (with 15cm error). Briefly, the ball sends a POLL, the anchor sends back a RESPONSE, and the ball responds with a FINAL packet. Using the two round trip times, and the corresponding turn-around delays, the time of flight is computed without requiring time synchronization between the devices (algorithm details in [21, 4]). Multiplied by the speed of light, this yields the ball's range. This is not our contribution since the Decawave platform offers the foundational capabilities.

Observe that UWB ranging is available from only 2 anchors (placed at the two wickets) and therefore inadequate to resolve the 3D location of the ball. Additional anchors cannot be placed on the ground since it will interfere with the motion of the ball and fielders, while placing anchors outside the field (90m away from the wickets) significantly degrades SNR and ranging accuracy. Fig.10 shows the intersections of the 2 anchor measurements, i.e., circles formed by the intersection of two spheres centered at the anchors. At a given time, the ball can lie on any point of a circle. Given that the initial position and velocity of the ball is unknown, many 3D trajectories can satisfy these constraints.

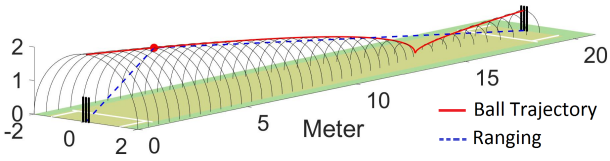


Figure 10: Intersections of ranging measurements leave one location dimension unresolved.

4.2 Mobility Constraints

We bring two mobility constraints to resolve the uncertainty: (1) physics of ball motion, and (2) opportunities from the ball's bouncing position.

(1) Physics of Ball Motion

Fig.11 shows a free-body diagram depicting the forces acting on the Cricket ball while in flight. Besides gravity, aerodynamic forces are acting on the ball. Briefly, the ball surface is smooth on one side of the seam and rough on the other (Cricket bowlers continue to polish the smooth side during the game). This disparity causes unbalanced air-flow, causing a side force. The speed of the ball can cause a slight air drag force. The magnitude and direction of the side forces depends on the seam orientation, surface roughness, and ball velocity. The drag and side force coefficients can be approximated as constants [14]. The side force can produce up to 1m of lateral deflection in trajectory.

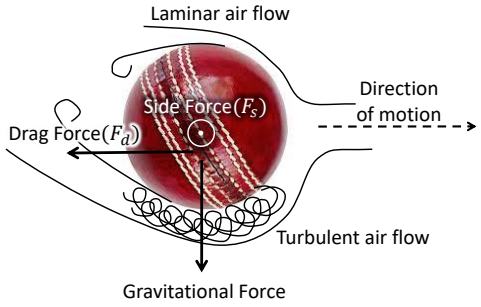


Figure 11: Unbalanced air-flow due to asymmetric smoothness on the ball's surface causes side force.

Under the above forces, ball motion follows a simple projectile path [14].

Fig.12(a) shows the extent to which the projectile model (without the aerodynamic forces) fits the ball's true tra-

jectory (derived from ViCon). The projectile was seeded by the ViCon-computed initial location and initial velocity. The median error is $1cm$ across 25 different throws of the ball, offering confidence on the usability of the model in indoor environments. The efficacy outdoors remains an open question.

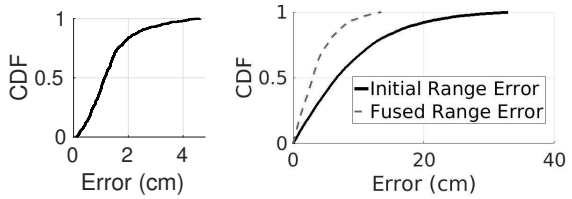


Figure 12: (a) Error between a parametric motion model and ground truth (derived from ViCon cameras). (b) Reduced ranging error after fusing UWB ranging with parameterized motion models.

(2) Bouncing Constraint

When the ball bounces before reaching the batsman (detectible from an accelerometer spike), the Z component of location – the ball's height – is 0. This resolves the uncertainty at a single point, i.e., in combination with UWB ranging, the ball's location can be computed *only* at this point. Thus, one point on the trajectory is "pinned down", shortlisting a smaller set of candidate trajectories. We can now fuse these physical constraints with the underdetermined system from Fig.10.

On the other hand, there might be cases where the ball does not bounce, resulting in slight degradation in accuracy. However, sensors are being embedded in bats as well; thus, the bat's location combined with the ball-bat contact time could serve as a virtual bounce, reducing uncertainties at a single point. This paper has not pursued such opportunities and leaves them to future work.

4.3 Fusing Range and Motion Constraints

Our goal is to model the trajectory as an error minimization problem. We denote the two anchor positions as $(x_{ia}, y_{ia}, z_{ia})\ \forall\ i \in \{1,2\}$. Also, we denote the initial location and initial velocity of the ball – at the point of release from the hand – as (x_o, y_o, z_o) and (v_x, v_y, v_z), respectively. Thus, at a given time t, the estimated location of the ball from simple physics models (without aerodynamics) is:

$$S_{xe}(t) = x_o + v_x t \tag{4}$$

$$S_{ye}(t) = y_o + v_y t \tag{5}$$

$$S_{ze}(t) = z_o + v_z t - 0.5gt^2 \tag{6}$$

Here, g is the acceleration due to gravity. Using this, the range from each anchor i is parameterized as:

$$R_{i,p}(t) = \sqrt{(S_{xe} - x_{ia})^2 + (S_{ye}(t) - y_{ia})^2 + (S_{ze}(t) - z_{ia})^2} \tag{7}$$

Once we have the range modeled, we design the error function, Err, as a difference of the parameter-modeled range and the measured range as follows.

$$\underset{6params}{\text{argmin}}\ \ Err = \sum_{i=1,2} \sum_{t} \{R_{i,p}(t) - R_{i,m}(t)\}^2 \tag{8}$$

This objective function is minimized using a gradient descent algorithm, however, since it is highly *non-convex*, multiple local maxima exist. We bound the search space based on 2 boundary conditions: (1) The Z coordinate of the bouncing location is zero. (2) The initial ball-release location is assumed to be within a $60cm^3$ cube, as a function bowler's height.

While this proved effective in eliminating many local maxima, Fig.12(b) shows that the median ranging error is $3cm$. However, translating range to location is affected by a phenomenon called *dilution of precision* (DoP) [26].

4.4 Dilution of Precision (DoP)

Ideally, the intersection of two UWB range measurements (i.e., two spheres centered at the anchors) is a circle – the ball should be at some point on this circle. In reality, ranging error causes the intersection of spheres to become 3D "tubes". Now, when the two spheres become nearly tangential to each other, say when the ball is near the middle of two anchors, the region of intersections becomes large. Fig.13(b) shows this effect. This is called DoP and seriously affects the location estimate of the ball (later we will see how DoP in Fig.13(c) affects the localization of the players). DoP is a fundamental problem that affects other trilateration applications like GPS.

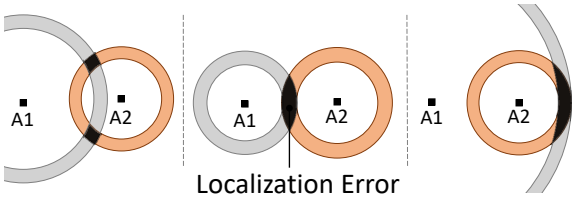


Figure 13: DoP introduced from ranging errors: (a) Lower DoP when ranging circles not tangential, (b) higher DoP when circles externally tangential, (c) max DoP when circles internally tangential.

Fig.14 shows the error variation as the ball moves in flight. The ball is released at time $t = 0$ and it reaches the batsman by the end of the flight. – clearly, the error

increases and is maximal near the middle of the flight. However, since the DoP can be modeled as a function of distance from the anchors, it should be possible to *weigh* the errors in the minimization function as follows:

$$\underset{6params}{\text{argmin}} \; Err = \sum_{i=1,2} \sum_{t} \{(R_{i,p}(t) - R_{i,m}(t)) \times \frac{1}{\sqrt{(DoP)}}\}^2 \tag{9}$$

This revised minimization function pays less importance to range measurements weighted by a large DoP. Results improve to a median of 16*cm* error.

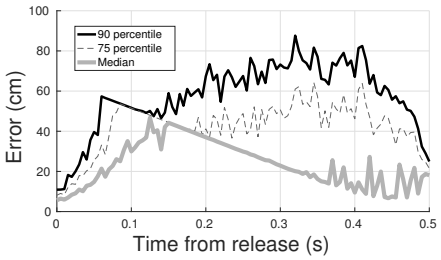


Figure 14: DoP aggravates error near the middle.

4.5 Exploiting Angle of Arrival (AoA)

The MIMO antennas at the anchors are capable of synchronized phase measurements of the incoming signal. Fig.15 shows how the phase difference ϕ is a function of the difference in signal path (p_1 and p_2), which is in turn related to AoA, θ. Thus, we have:

$$d\cos(\theta)\frac{2\pi}{\lambda} = \phi \tag{10}$$

$$\cos(AoA) = \cos(\theta) = \frac{\phi\lambda}{2\pi d} \tag{11}$$

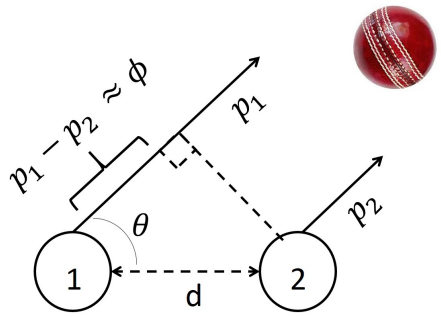


Figure 15: AoA is derived from phase differences.

We employ a MIMO receiver only on the bowler side (the other anchor cannot be utilized since it gets significantly interfered by the batsman, corrupting phase measurements). Now, for this single anchor, say the antennas are separated along the x-axis; then the AoA can be expressed in terms of ball location, anchor locations, and measured ranges:

$$\cos(\theta) = \frac{S_x - x_{ia}}{R_{ia}} \tag{12}$$

$$S_{x,aoa} = \cos(\theta)R_{ia} + x_{ia} \tag{13}$$

Thus, it is possible to refine the previous estimates of the trajectory by including AoA in the error function. Finally, DoP problems arise with AoA too – as the ball travels further away from the anchor, the location error increases for a small $\Delta\theta$ error in AoA. In fact, the error is $R\Delta\theta$, where R is the ball's range.

4.6 Exploiting Antenna Separation

It is clear from Equation 11 that the AoA error is a function of antenna separation d – higher antenna separation will decrease the error in measurement of $\cos\theta$ (AoA). However, with antenna separation higher than wavelength λ, the phase wraps and introduces ambiguity in AoA estimation – called *integer ambiguity*. For unambiguous AoA measurements, $d\cos\theta < \frac{\lambda}{2}$ or $d < \frac{\lambda}{2}$. Fig.16(a) shows a common case of unambiguous AoA measurement during a ball throw. Evidently, AoA is heavily corrupted from spinning antenna orientation and polarization.

To mitigate the noise, we increase the antenna separation d. However, when $d > \frac{\lambda}{2}$, the ambiguous AoA measurements are indicated in the equation below.

$$d\cos(\theta) = \frac{\phi\lambda}{2\pi} + N\lambda \tag{14}$$

$$\cos(\theta) = \frac{\phi\lambda}{2\pi d} + N\frac{\lambda}{d} \tag{15}$$

The AoA is not only a function of the phase difference ϕ, but also a function of the unknown integer ambiguity N. Fortunately, the smooth trajectory of the ball provides an opportunity for tracking the integer ambiguity across measurements, thereby any wrap around can be detected and accounted for. Fig.16(b) shows a common case of AoA measurement (known integer ambiguity) for a ball throw after increasing the antenna separation to 18 cm – 2.5 times the wavelength. Evidently, the noise is much lower, offering an additional opportunity.

4.7 Fusion of AoA with Ranging/Physics

In order to fuse, AoA, we need to firstly resolve the integer ambiguity. Our technique to resolve this is simple. At any point during the gradient search algorithm, we obtain an estimated AoA from current set of parameters. We simply resolve the integer ambiguity by substituting the currently estimated AoA in Equation 15. Incorrect ambiguity resolution would automatically explode the error function because of mismatch with range measurements.

(For example, if the integer ambiguity resolution is incorrect even by a single integer, that would introduce a median mismatch of 7.5cm (one wavelength) between inferred and measured ranges). With integer ambiguity resolution, we are ready to update the objective function Err, with AoA fusion.

$$\underset{6params}{\text{argmin}}\ Err = \sum_{i=1,2}\sum_{t}\{(R_{i,p}(t) - R_{i,m}(t)) \times \frac{1}{\sqrt{(DoP)}}\}^2 + \sum_{t}\{\frac{(S_{x,p}(t) - S_{x,aoa}(t))}{R_{aoa,p}\Delta\theta}\}^2 \quad (16)$$

where $S_{x,aoa}(t)$ is drawn from Equation 12. The $R_{aoa,p}\Delta\theta$ (R_{aoa} denotes range from AoA anchor, $\Delta\theta$ is AoA noise) factor decreases the weight for AoA measurements taken far away from the AoA Anchor.

To summarize, iBall incorporates noisy ranging and AoA measurements from UWB Anchors with physics based motion model to track the ball trajectory. Results are presented in Section.6.

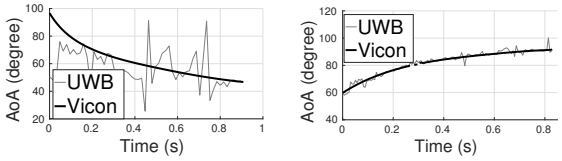


Figure 16: Improved AoA with antenna separation.

5 System Design: Player Tracking

iBall aims to track the movement of players in the field. Assuming clip-on UWB devices on the players, the ball ranging techniques should apply directly; in fact, since players are on the 2D ground, the tracking should be feasible with 2-anchor ranging alone. However, a different form of DoP emerges: when the two lines joining the player and the two anchors tend to get collinear, the ranging rings around the anchors begin to exhibit larger overlapping areas (see Fig.13(c)). Fig.17 shows simulations of DoP on a real-sized Cricket ground. As the player moves closer to the X axis (i.e., higher collinearity), the 90 percentile uncertainty of estimated location increases to $15m$, in contrast to $1m$ when the player is perpendicular to the anchors. The effect is worse with higher distance from anchors.

5.1 DoP Suppression through Filtering

To cope with DoP degradations, we apply Kalman Filtering (KF) to player tracking. The basic idea is to detect (from the accelerometer) that a player has started running, and combine the motion model of a human-run with the UWB ranging estimates. For the human-run model, we assume the velocity to be piecewise constant

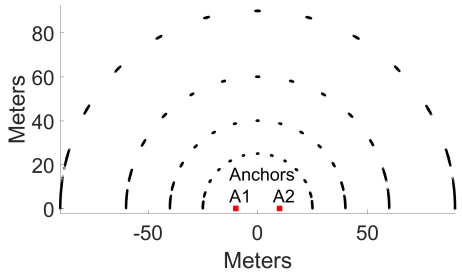


Figure 17: Simulation of estimated player location.

in short time scales (second). The velocity is periodically updated using the recent KF estimates, thereby accounting for changes in the human's run patterns. Section 6 will show results from real experiments on a large field. iBall applies the same techniques to the ball, which can also be tracked after it has been hit by the batsman. We evaluate the overall system next.

6 Evaluation

Our experiments were performed in a $10x10x4m^3$ indoor space with 8 ViCon IR cameras installed on the ceiling for ground truth (Fig.18(a)). Fig.18(b) shows IR markers pasted on the ball – this enables location and orientation tracking accuracies of $0.1mm$ and 0.2°, respectively. The authors pretended to be Cricket players and threw the ball at various speeds and spins (for a total of 100 throws). A batsman was realistically positioned to create signal blockage between the ball and the anchor. The Intel curie chip provides IMU data at $70Hz$, while the anchors perform ranging/AoA at $150Hz$.

Metrics: At any given time t during the flight of the ball, ViCon camera provides the true orientation of the ball, say $C(t)$, while iBall generates an estimated orientation, say $E(t)$. The **Orientation Error (ORE)** is essentially the minimum rotation that must be applied to $E(t)$ to align with $C(t)$. We measure this error across different values of t and plot the CDF. We also plot the angle difference between the rotation axis and the seam plane, called **Rotation Axis Error (AXE)** and **Seam Plane Error (PLE)**. Now, the true angular velocity of the ball, C_ω can be computed as $\frac{C(t_2)C(t_1)^{-1}}{t_2-t_1}$ (note that difference in orientation matrices is computed through inverse functions). When multiplied by the time of the flight, the result is the total cumulative angle truly rotated by the ball. We compute the same from $E(t)$ and ultimately compute the difference in the **Cumulative Angle Error (CUE)**. To understand the impact of higher spin, we also report the orientation error (OE) for varying angular velocity. For trajectory, the metrics are simpler, namely **Location Error (LOE)** and **Speed Error (SPE)** reported against various parameters.

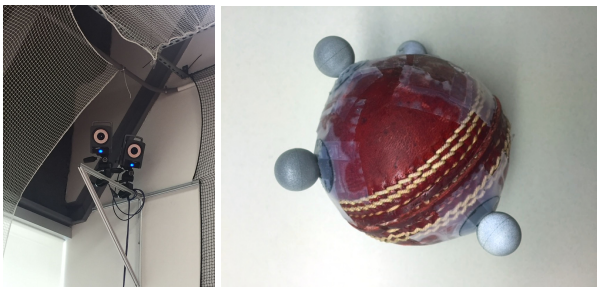


Figure 18: (a) IR based ViCon cameras at the ceiling. (b) A cricket ball instrumented with IR markers.

6.1 Performance of Spin Tracking

(1) Cumulative Angle Error (CUE): Fig.19 reports the CUE for each of the 50 spin throws – the results are sorted in ascending order of cumulative angles. A Y axis value of 4000 implies that the ball has rotated $\frac{4000}{360}$, which is 11.1 cycles in air at the end of the flight. Evidently, iBall performs close to the Vicon ground truth for almost every throw. More importantly, unlike gyroscopes (which suffer from drift), the magnetometer does not accumulate error over time (since it measures the absolute North vector at every sample). This is a promising result and a valuable primitive for various types of ball analytics.

Error in estimated angular velocity (not shown) follows the same trend as Fig.19(a), since it is simply the cumulative rotation divided by flight time. Across 50 experiments, we observe median angular velocity error of 1.0% and a maximum error of 3.9%.

(2) Overall Orientation Error (ORE): Fig.19(b) reports the CDF of ORE across all 50 throws – the median is 11.2°. We also break down this error into local rotation axis error (AXE) and seam plane orientation error (PLE). We are especially interested in these two because accurately controlling rotation axis and seam plane is critical in maintaining the stability of the ball in the air. Results show a median AXE of $< 5°$ and a median PLE of $< 8°$, while the 90^{th} percentile remains $< 20°$.

(3) Impact of High Spin: Fig.19(c) reports the impact of higher spin on ORE. The accuracy slightly degrades as the angular velocity increases. This is because, at higher angular velocity, our estimation of global rotation axis $\hat{R}^G$ is less accurate, degrading ORE. However, since the flight time is short, the accuracy degradation is marginal.

6.2 Performance of Trajectory Tracking

(1) Overall Location Error (LOE): Fig.20(a) quantifies the location error (LOE) across 50 different throws – the median error is $8cm$. We also report the errors on each of the directions: Y in the direction of the throw, Z being vertically upwards, and X is perpendicular to Y and Z. The median X, Y, and Z axes errors are $4.5cm$, $3.4cm$ and $2.39cm$ respectively. The X axis errors are maximum due to DoP effects, however, AoA lowers it to a reasonable value.

(3) Does LOE Accumulate at the End of the Flight? Fig.20(b) shows the median, 25^{th}, and 75^{th} percentile error for different positions of the ball during the flight. Importantly, since we solve a global error minimization problem, the error does not accumulate. Still, the initial positions have higher accuracy compared to the end of the flight because AoA computed from the bowler–side anchor exhibits significantly less error. The degradation is still modest, with a median end-flight LOE of $15cm$.

(3) Speed Error (SPE) and Impact of Speed: iBall computes velocity estimates – Fig.20(c) shows a median speed error (SPE) of $0.4m/s$. Upon discussions with domain experts, we gather that this level of accuracy is valuable for coaching and analytics. Fig.21(a) decomposes the overall LOE results into different speed buckets. Evidently, the accuracy does not degrade at higher speed regimes (the maximum speed we could achieve in our experiments was $22m/s$). Of course, this is indoors and wind effects are minimal, if any.

(4) Trajectory Extrapolation: The ball sometimes hits the leg of the batsman. An important question in Cricket is: *would the ball hit the wicket if the batsman was not in the way?* This is called *leg before wickets (LBW)*. For LBW decisions, its valuable to be able to predict (or extrapolate) the trajectory of the ball. The International Cricket Association has declared $10cm$ as the minimum tolerable LOE for LBW decisions. Fig.21(b) shows the trajectory prediction error with iBall. While the 3D LOE is $22cm$, the x-axis error is smaller ($9.9cm$), indicating the feasibility of using iBall for LBW decisions.

6.3 Performance of Player Tracking

Fig.21(c) shows LOE when the player is running at different parts of the playground – each line in the graph corresponds to the angle made by the lines joining the player and the two anchors. This also represents the LOE of the ball after it has been hit. We experimented in a real playground with a user running in a precise peripheral circle of $89m$ radius. At low angles (i.e., running almost collinear with the two anchors), the 90^{th} percentile error can be as large as $3.5m$. Our Kalman Filter based approach reduces this LOE to $2.6m$, while at higher angles, the LOE is already less than $50cm$.

7 Discussion and Future Work

(1) **Scaling to greater speed and spin:** Our maximum throw speeds were limited by our own abilities. Perhaps

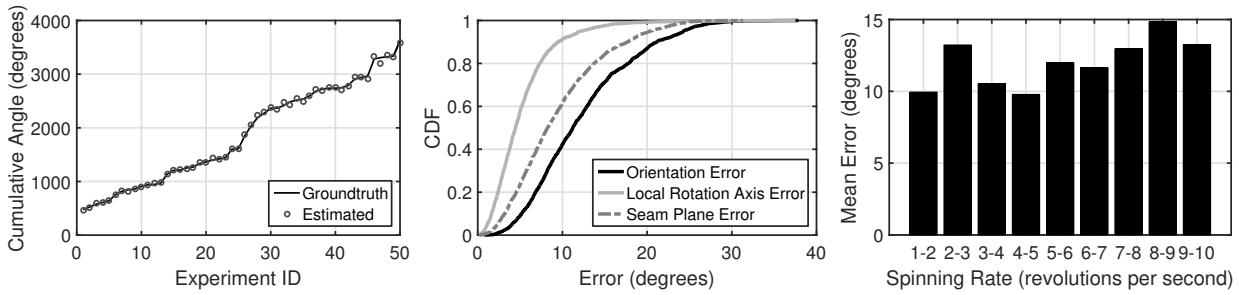


Figure 19: (a) Cumulative angle error (CUE) across different experiments. (b) CDF of orientation error (ORE). (c) Average orientation tracking error (ORE) under different spinning rate.

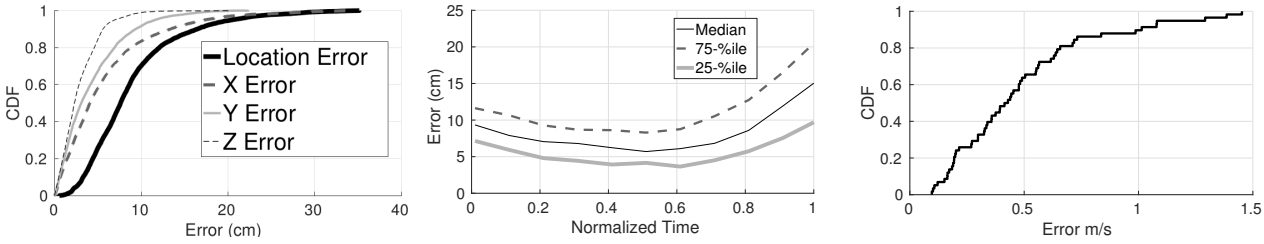


Figure 20: (a) CDF of location tracking errors. (b) iBall's location error degrades slightly toward the end of ball flight. (c) CDF of ball speed error.

a bowling machine would serve as a better experimentation platform. For spin, limitations arose from ViCon – we observed increasing jitters and discontinuities in the ViCon data for spins above $12rps$. We are exploring alternative ground-truth estimation techniques at such spin regimes.

(2) **Indoor experiments:** We need experimentation under outdoor aerodynamic effects – the reported results may have been favorable in its absence. The lack of an outdoor ground truth system has been the bottleneck so far – we are exploring alternatives.

(3) **Multipath:** On the other hand, indoor environments may have affected the AoA estimates as well. In an outdoor setting, wireless multipath is expected to be less pronounced, potentially offering better reliability with AoA estimation and fusion.

(4) **Generalizing to other sports:** We believe iBall's techniques can be extended to other sports with domain specific modifications. For example, stitches on a baseball induce different aerodynamic effects, however, these differences can be modeled and incorporated into iBall. Such models are also available for golf and tennis balls, allowing them to be suitably "plugged" into our framework. Finally, iBall's techniques may extend to hollow balls like soccer and basketball. Adidas micoach [2] has designed a soccer ball with multiple suspended sensors within the ball. This can potentially offer more information to iBall's optimization engine.

(5) **Smart ball weight distribution:** Needless to say, our ball prototype is not ready for real use – the embedded sensor is not optimized to preserve the homogeneous mass distribution inside the ball. This may have led to some biases in the trajectory and spin results, although we believe it is marginal. In the longer run, mechanical engineering experts from D2M [3] have corroborated that a near ideal weight distribution (with impact tolerance) is feasible. The opportunities arise from smaller spatial footprint of the device, eliminating the battery (by harvesting energy from the ball's spin), and combining the IMU and the radio in a single smaller chip. Validating our results on such a professionally manufactured ball remains a part of future work.

(6) **Enhancing accuracy:** We believe there is room to improve the location and spin tracking accuracy of iBall. For example, it should be possible to jointly estimate the location and rotation instead of treating them as separate modules. The accelerometer can measure a combination of centripetal force (rotation) and linear acceleration. Similarly, the UWB ranging measurements contain a few bits of information about the orientation since the radios cannot be precisely placed at the center of mass of the ball. Such a coupling suggests that joint estimation of can improve the accuracy of both. Hardware opportunities that leverage dual carrier UWB receivers can further decrease errors in AoA, perhaps at a slight increase in hardware complexity.

(6) **Battery life and connectivity:** The current ball pro-

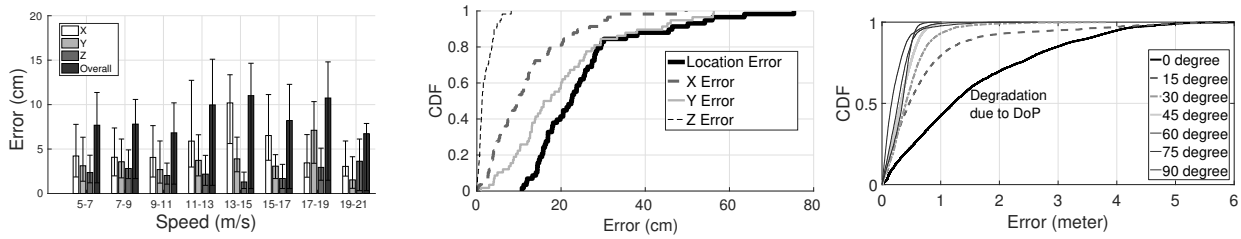


Figure 21: (a) LOE across ball speeds. (b) Prediction accuracy sufficient for LBW. (c) Player location error.

totype allows a battery life of $\approx 75-90$ minutes between recharges, permitting short training sessions. In future, perhaps wireless charging will mitigate this problem; perhaps fast rotation will automatically scavenge energy. Additional energy optimizations could be made in the compute as well as the communication pipeline, i.e., when the ball sends the sensor data to the anchor. As one simple example, the anchors could perhaps beamform towards the ball to minimize the ball's transmission energy. Since the anchor is equipped with a large battery, such asymmetric designs should be viable.

8 Related Work

Embedded IMU: Authors in [19, 17, 18, 32] embed IMUs in a Cricket ball and is perhaps closest to our work. However, these (brief) papers report basic features such as angular velocity, time of flight, etc. These features are directly available from the sensors and do not address the actual metrics of interest to the players/coaches. Authors in [16] also embed IMUs but focus mainly on the design and packaging of the ball for high impact. [24] explores spin-analytics in the context of a Bowling ball, however, due to low spin-rates and contact with the floor, accelerometers and gyroscopes are readily usable. This simplifies the problem in contrast to Baseball and Cricket.

Wearables, Cameras, and Sports Analytics: Several startups like Zepp, MiCoach, and Ball are extracting motion patterns from wearables. Smart sensor shoes have been proposed for analyzing soccer shots in [47], however, these are essentially classification problems. Hawk-Eye [7] is perhaps the most popular and expensive camera based tracker officially adopted in Cricket, Tennis, etc. Hot Spot [10] is a popular IR technology used to determine contact points between ball and players. Video analytics efforts in [23, 37, 46] are processing video feeds to learn/predict game strategies. While creative, the projects are addressing a different set of problems.

Localization and Motion Tracking: Rich literature in indoor localization [13, 45, 15, 43, 33, 42, 36, 41, 25] has mostly focused on human motion. Under sparse WiFi infrastructure and high ball speeds, such techniques are inadequate. UWB based ToF ranging [31] report 10cm accuracy for static objects. We build on this technique but fuse with AoA, motion models, and DoP constraints, to cope with real-world challenges. On a similar note, inertial sensor based tracking have mostly been performed on humans, robots and drones [29, 30, 48, 28, 27, 35]. However, unlike iBall, none of these works address the space of freely falling objects. While work in [38] tracks ballistic missiles, the granularity of tracking is different both in time and space. iBall entails much finer granularities of tracking and appropriately formulates a global optimization problem for better accuracy unlike filtering techniques in [38].

9 Conclusion

This paper develops techniques for tracking the 3D trajectory and spin parameters of a cricket ball. The core problem is rooted in motion tracking techniques, however, the sporting applications (and Cricket in this case) presents unique challenges and opportunities. Through fusion of wireless ranging, models of free-falling objects, and angle of arrival estimates, we formulate and solve error minimization problems. Results are promising and we expect our techniques to generalize to other sports. Our ongoing work is in pursuit of baseball and frisbee.

10 Acknowledgments

We sincerely thank our shepherd Dr. Anirudh Badam and the anonymous reviewers for their valuable feedback. We are also grateful to NSF (CNS - 1423455) for partially funding the research. We acknowledge the support of various teams in bringing the system together. 1) Hardware and Software engineers at Intel[11] for helping on various design aspects of the embedded sensor hardware and software. 2) D2M[3] for ball design and prototyping. 3) The International Cricket Council (ICC) [12] and Narayan Sundararajan for providing domain expertise on the game of cricket and the business of sports analytics.

References

[1] `http://www.jamesgibbard.co.uk/electronics/bluetoothcontrolledledflashingfrisbee`.

[2] adidas micoach smart ball. http://www.adidas.com/us/micoach-smart-ball/G83963.html.

[3] D2m. http://d2m-inc.com/.

[4] Decawave.

[5] Firstv1sion. http://www.firstv1sion.com/en/product/.

[6] Goal-line technology. https://en.wikipedia.org/wiki/Goal-line_technology.

[7] Hawk-eye. https://en.wikipedia.org/wiki/Hawk-Eye/.

[8] High range gyroscopes. http://www.analog.com/en/products/mems/gyroscopes.html.

[9] High speed camera. http://www.untamedscience.com/filmmaking/advanced-filmmaking/high-speed-video-slow-motion/.

[10] Hot spot. https://en.wikipedia.org/wiki/Hot_Spot_(cricket)/.

[11] Intel, data center solutions, iot and pc innovation. http://www.intel.com.

[12] International cricket council. http://www.icc-cricket.com.

[13] BAHL, P., AND PADMANABHAN, V. N. Radar: An in-building rf-based user location and tracking system. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* (2000), vol. 2, Ieee, pp. 775–784.

[14] BAKER, C. A calculation of cricket ball trajectories. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science 224*, 9 (2010), 1947–1958.

[15] CHINTALAPUDI, K., PADMANABHA IYER, A., AND PADMANABHAN, V. N. Indoor localization without the pain. In *Proceedings of the sixteenth annual international conference on Mobile computing and networking* (2010), ACM, pp. 173–184.

[16] FUSS, F., FERDINANDS, R., DOLJIN, B., AND BEACH, A. Development of a smart cricket ball and advanced performance analysis of spin bowling. In *ICSST 2014: Advanced Technologies in Modern Day Sports* (2014), Institute for Sports Research (ISR), pp. 588–595.

[17] FUSS, F. K., LYTHGO, N., SMITH, R. M., BENSON, A. C., AND GORDON, B. Identification of key performance parameters during off-spin bowling with a smart cricket ball. *Sports Technology 4*, 3-4 (2011), 159–163.

[18] FUSS, F. K., AND SMITH, R. M. Accuracy performance parameters of seam bowling, measured with a smart cricket ball. *Procedia Engineering 72* (2014), 435–440.

[19] FUSS, F. K., SMITH, R. M., AND SUBIC, A. Determination of spin rate and axes with an instrumented cricket ball. *Procedia Engineering 34* (2012), 128–133.

[20] GUPTA, S., MORRIS, D., PATEL, S., AND TAN, D. Soundwave: using the doppler effect to sense gestures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2012), ACM, pp. 1911–1914.

[21] HACH, R. Symmetric double sided two-way ranging. *IEEE P802 15* (2005), 802–15.

[22] HAHNEL, D., BURGARD, W., FOX, D., AND THRUN, S. An efficient fastslam algorithm for generating maps of large-scale cyclic environments from raw laser range measurements. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on* (2003), vol. 1, IEEE, pp. 206–211.

[23] HALVORSEN, P., SÆGROV, S., MORTENSEN, A., KRISTENSEN, D. K., EICHHORN, A., STENHAUG, M., DAHL, S., STENSLAND, H. K., GADDAM, V. R., GRIWODZ, C., ET AL. Bagadus: an integrated system for arena sports analytics: a soccer case study. In *Proceedings of the 4th ACM Multimedia Systems Conference* (2013), ACM, pp. 48–59.

[24] KING, K., PERKINS, N. C., CHURCHILL, H., MCGINNIS, R., DOSS, R., AND HICKLAND, R. Bowling ball dynamics revealed by miniature wireless mems inertial measurement unit. *Sports Engineering 13*, 2 (2011), 95–104.

[25] KUMAR, S., GIL, S., KATABI, D., AND RUS, D. Accurate indoor localization with zero start-up cost. In *Proceedings of the 20th annual international conference on Mobile computing and networking* (2014), ACM, pp. 483–494.

[26] LANGLEY, R. B. Dilution of precision. *GPS world 10*, 5 (1999), 52–59.

[27] LEFFERTS, E. J., MARKLEY, F. L., AND SHUSTER, M. D. Kalman filtering for spacecraft attitude estimation. *Journal of Guidance, Control, and Dynamics 5*, 5 (1982), 417–429.

[28] LIANG, W. Y., MIAO, W. T., HONG, L. J., LEI, X. C., AND CHEN, Z. Attitude estimation for small helicopter using extended kalman filter. In *Robotics, Automation and Mechatronics, 2008 IEEE Conference on* (2008), IEEE, pp. 577–581.

[29] MADGWICK, S. An efficient orientation filter for inertial and inertial/magnetic sensor arrays. *Report x-io and University of Bristol (UK)* (2010).

[30] MAHONY, R., HAMEL, T., AND PFLIMLIN, J.-M. Nonlinear complementary filters on the special orthogonal group. *IEEE Transactions on Automatic Control 53*, 5 (2008), 1203–1218.

[31] MCELROY, C., NEIRYNCK, D., AND MCLAUGHLIN, M. Comparison of wireless clock synchronization algorithms for indoor location systems. In *2014 IEEE International Conference on Communications Workshops (ICC)* (2014), IEEE, pp. 157–162.

[32] MCGINNIS, R. S., AND PERKINS, N. C. A highly miniaturized, wireless inertial measurement unit for characterizing the dynamics of pitched baseballs and softballs. *Sensors 12*, 9 (2012), 11933–11945.

[33] NICULESCU, D., AND NATH, B. Ad hoc positioning system (aps) using aoa. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies* (2003), vol. 3, Ieee, pp. 1734–1743.

[34] NIKE. Footwear having sensor system. Patent US 8676541 B2.

[35] PFLIMLIN, J. M., HAMEL, T., AND SOUÈRES, P. Nonlinear attitude and gyroscope's bias estimation for a vtol uav. *International Journal of Systems Science 38*, 3 (2007), 197–210.

[36] RAI, A., CHINTALAPUDI, K. K., PADMANABHAN, V. N., AND SEN, R. Zee: zero-effort crowdsourcing for indoor localization. In *Proceedings of the 18th annual international conference on Mobile computing and networking* (2012), ACM, pp. 293–304.

[37] SEO, Y., CHOI, S., KIM, H., AND HONG, K.-S. Where are the ball and players? soccer game analysis with color-based tracking and image mosaick. In *International Conference on Image Analysis and Processing* (1997), Springer, pp. 196–203.

[38] SIOURIS, G. M., CHEN, G., AND WANG, J. Tracking an incoming ballistic missile using an extended interval kalman filter. *IEEE Transactions on Aerospace and Electronic Systems 33*, 1 (1997), 232–240.

[39] SWANSON, E. Geometric dilution of precision. *Navigation 25*, 4 (1978), 425–429.

[40] TSAI, N.-C., AND SUE, C.-Y. Stability and resonance of micromachined gyroscope under nonlinearity effects. *Nonlinear Dynamics 56*, 4 (2009), 369–379.

[41] VASISHT, D., KUMAR, S., AND KATABI, D. Decimeter-level localization with a single wifi access point. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (2016), pp. 165–178.

[42] WANG, H., SEN, S., ELGOHARY, A., FARID, M., YOUSSEF, M., AND CHOUDHURY, R. R. No need to war-drive: unsupervised indoor localization. In *Proceedings of the 10th international conference on Mobile systems, applications, and services* (2012), ACM, pp. 197–210.

[43] XIONG, J., AND JAMIESON, K. Arraytrack: a fine-grained indoor location system. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013), pp. 71–84.

[44] YANG, L., CHEN, Y., LI, X.-Y., XIAO, C., LI, M., AND LIU, Y. Tagoram: Real-time tracking of mobile rfid tags to high precision using cots devices. In *Proceedings of the 20th annual international conference on Mobile computing and networking* (2014), ACM, pp. 237–248.

[45] YOUSSEF, M., AND AGRAWALA, A. The horus wlan location determination system. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services* (2005), ACM, pp. 205–218.

[46] YU, X., XU, C., LEONG, H. W., TIAN, Q., TANG, Q., AND WAN, K. W. Trajectory-based ball detection and tracking with applications to semantic analysis of broadcast soccer video. In *Proceedings of the eleventh ACM international conference on Multimedia* (2003), ACM, pp. 11–20.

[47] ZHOU, B., KOERGER, H., WIRTH, M., ZWICK, C., MARTINDALE, C., CRUZ, H., ESKOFIER, B., AND LUKOWICZ, P. Smart soccer shoe: monitoring foot-ball interaction with shoe integrated textile pressure sensor matrix. In *Proceedings of the 2016 ACM International Symposium on Wearable Computers* (2016), ACM, pp. 64–71.

[48] ZHOU, P., LI, M., AND SHEN, G. Use it free: instantly knowing your phone attitude. In *Proceedings of the 20th annual international conference on Mobile computing and networking* (2014), ACM, pp. 605–616.

FarmBeats: An IoT Platform for Data-Driven Agriculture

Deepak Vasisht[1,2], Zerina Kapetanovic[1,3], Jong-ho Won[1,4], Xinxin Jin[1,5], Ranveer Chandra[1], Ashish Kapoor[1], Sudipta N. Sinha[1], Madhusudhan Sudarshan[1], Sean Stratman[6]

[1]Microsoft, [2]MIT, [3]University of Washington, [4]Purdue University, [5]UCSD, [6]Dancing Crow Farm

Abstract – Data-driven techniques help boost agricultural productivity by increasing yields, reducing losses and cutting down input costs. However, these techniques have seen sparse adoption owing to high costs of manual data collection and limited connectivity solutions. In this paper, we present FarmBeats, an end-to-end IoT platform for agriculture that enables seamless data collection from various sensors, cameras and drones. FarmBeats's system design that explicitly accounts for weather-related power and Internet outages has enabled six month long deployments in two US farms.

1 Introduction

The demand for food is expected to double by 2050, primarily fueled by an increase in population and upward social mobility [58]. Achieving this increase in food production is even more challenging because of receding water levels, climate change and shrinking amount of arable land. According to International Food Policy Research Institute, data-driven techniques can help us achieve this goal by increasing farm productivity by as much as 67% by 2050 and cutting down agricultural losses [20].

In fact, field trials have shown that techniques that use sensor measurements to vary water input across the farm at a fine granularity (precision irrigation) can increase farm productivity by as much as 45% while reducing the water intake by 35% [3]. Similar techniques to vary other farm inputs like seeds, soil nutrients, etc. have proven to be beneficial [25, 37]. More recently, the advent of aerial imagery systems, such as drones, has enabled farmers to get richer sensor data from the farms. Drones can help farmers map their fields, monitor crop canopy remotely and check for anomalies. Over time, all this data can indicate useful practices in farms and make suggestions based on previous crop cycles; resulting in higher yields, lower inputs and less environmental impact.

While these techniques for agriculture have shown promising results, their adoption is limited to less than 20 percent farmers owing to the high cost of manual sensor data collection (according to US Department of Agriculture [30]). Automating sensor data collection requires establishing network connection to these sensors. However, existing connectivity solutions [11, 18] require a cellular data logger to be attached to each sensor (see Table 1 for a detailed comparison). These loggers cost around $1000 each in equipment cost with over $100 in subscription fee. Further, they are limited in the amount of data that they can send to few kilobytes per day. Clearly, these solutions do not scale up for large farms and cannot support high bandwidth sensors like cameras and drones, which rely on sending all their data to the cloud for processing [10,49]. This situation is further worsened by the fact that farms typically have limited cellular coverage [24] and are prone to weather-based Internet outages.

In this paper, we present FarmBeats, an end-to-end IoT platform for data-driven agriculture, that enables seamless data collection from various sensor types, i.e., cameras, drones and soil sensors, with very different bandwidth constraints. FarmBeats can ensure system availability even in the face of power and Internet outages caused by bad weather; scenarios that are fairly common for a farm. Further, FarmBeats enables cloud connectivity for the sensor data to enable persistent storage as well as long-term or cross-farm analytics. We have deployed FarmBeats in two farms in the US over a period of six months and used FarmBeats to enable three applications for the farmer: precision agriculture, monitoring temperature and humidity in food storage, and monitoring animal shelters. In designing FarmBeats, we solve three key challenges.

First, to enable connectivity within the farm, FarmBeats leverages recent work in unlicensed TV White Spaces (TVWS) [6, 16, 44] to setup a high bandwidth link from the farmer's home Internet connection to an IoT base station on the farm. Sensors, cameras and drones can connect to this base station over a Wi-Fi frontend. This ensures high bandwidth connectivity within the farm. However, due to the lack of power on the farm, the base station is powered by battery-backed solar power which suffers from power unreliability depending on weather conditions. As shown in past work [22, 51], cloudy weather can reduce solar power output significantly and drain the batteries of the base station to shut it down. To solve this problem, FarmBeats uses a novel weather-aware IoT base station design. Specifically, it uses weather forecasts to appropriately duty cycle different components of the base station. To the best of our knowledge, this is the first weather-aware IoT base station design.

Second, Internet connection to the farm is typically weak making it challenging to ship high bandwidth drone videos (multiple GBs) to the cloud. Furthermore, farms are prone to weather-related network outages that last weeks. Such system unavailability impedes a farmer's

ability to take adequate preventive actions, do UAV inspections and leads to loss of valuable sensor data. Thus, FarmBeats uses a Gateway based design, wherein a PC at the farmer's home serves as a gateway for the farm data. The FarmBeats Gateway serves two purposes: a) it performs significant computation locally on the farm data to consolidate it into summaries that can be shipped to the cloud for long-term and cross-farm analytics, and b) the gateway is capable of independent operation to handle periods of network outage, thus leading to continuous availability for the farmer.

Finally, while drones are one of the most exciting farm sensors today, they suffer from poor battery life. Getting aerial imagery for a farm requires multiple drone flights and a long wait time in between when the batteries are being charged. We use the fact that farms are typically very windy, since they are open spaces. Thus, we incorporate a novel path planning algorithm in the FarmBeats gateway, that leverages wind to help the drone accelerate and decelerate, thereby conserving battery. This algorithm is motivated by how sailors use winds to navigate sailboats.

We use the FarmBeats system to enable precision agriculture applications on two farms: one in Washington state and the other in upstate New York. While traditional farming treats the farm as a homogeneous piece of land, precision agriculture adapts the farm inputs over different parts of the farm depending on the requirement. Precision agriculture techniques require a precision map with information about each location in the farm, for example, the soil temperature, soil moisture, nutrient levels, etc. To construct this precision map, existing solutions for precision agriculture require a dense deployment of in-ground sensors [30]. A dense deployment of sensors becomes expensive (as well as cumbersome to manage) as the size of the farm grows. Unless these sensors are deployed densely within a farm, the estimated precision map can be very inaccurate, as we show in Section 7. Since FarmBeats's gateway has access to both the drone videos and sensor data, it enables a novel low-cost mechanism that uses drone videos in combination with sparse ground sensors to generate precision maps for the farm. To the best of our knowledge, this is the first system that can combine the temporal data from sensors, with the spatial data from drones to construct an instantaneous precision map of the farm, such as the one in Figure 5.

Beyond FarmBeats's application in precision agriculture, farmers have so far used FarmBeats for two other applications. First, the farmers have been using FarmBeats to monitor temperature and humidity in storage spaces to ensure that the produce does not go bad. Second, the farmers have plugged in cameras at different locations, to monitor cow sheds, selling stations etc[1].

[1]Supplementary Material includes detailed description of FarmBeats applications and usage.

Contributions: To summarize, FarmBeats makes the following key contributions:

- **Long-term large scale deployment:** Our deployments have run over 6 months in each of the farms and collected over 10 million sensor measurements, 1 million camera images and 100 drone videos
- **Novel Weather-Aware IoT Base Station Design:** Adding weather awareness into the IoT base station reduced the base station down time to zero as opposed to greater than 30% downtime during the same month in the previous year in an earlier version of our deployment
- **Novel Inference Techniques for Compression of Aerial Imagery Data:** FarmBeats's gateway achieved a median compression of 1000 times from an aerial drone video to the sensor summaries sent to the cloud. Further, the gateway remained available even when the Internet connectivity to the farm faced a week-long outage
- **Wind-Assisted Drone Flight Planning Algorithm:** FarmBeats's flight planning algorithm improves the area covered by a single drone flight by 30%

2 IoT Platform: Objectives

In building FarmBeats, we target the following goals:

- **Availability:** The platform should have negligible downtime. When there is an outage (for example, due to power or network failure), data collection from the sensors should not stop and the platform should continue to deliver services to the farmers.
- **Capacity:** It should support sensors with widely varying requirements: pH sensors reporting few bytes of data to drones sending gigabytes of video. Similarly, the system should be capable of supporting end-user applications with varying needs: from a precision irrigation application that needs the latest sensor data for the entire farm to a crop suggestion application that needs just high level productivity data but across several growing seasons
- **Cloud Connectivity:** Several farming applications, such as crop cycle prediction, seeding suggestions, farming practice advisory, etc. rely on long term data analytics. Besides, a farmer may want to access some applications even when he is not on the farm. Thus, the IoT platform must enable pushing data to the cloud.
- **Data Freshness:** Stale sensor data from the farm can make applications suggest incorrect courses of action to the farmer. Gaps in historical data can also cause applications to misbehave. Moreover, stale data leads to bad user experience. Thus, the platform must strive to maintain maximum data freshness.

3 The FarmBeats IoT Platform

While these objectives have been fairly successfully achieved by home IoT platforms like Amazon Echo, achieving these objectives in an agricultural setting introduces several challenges for two main reasons: access and environmental variability. As discussed before (and

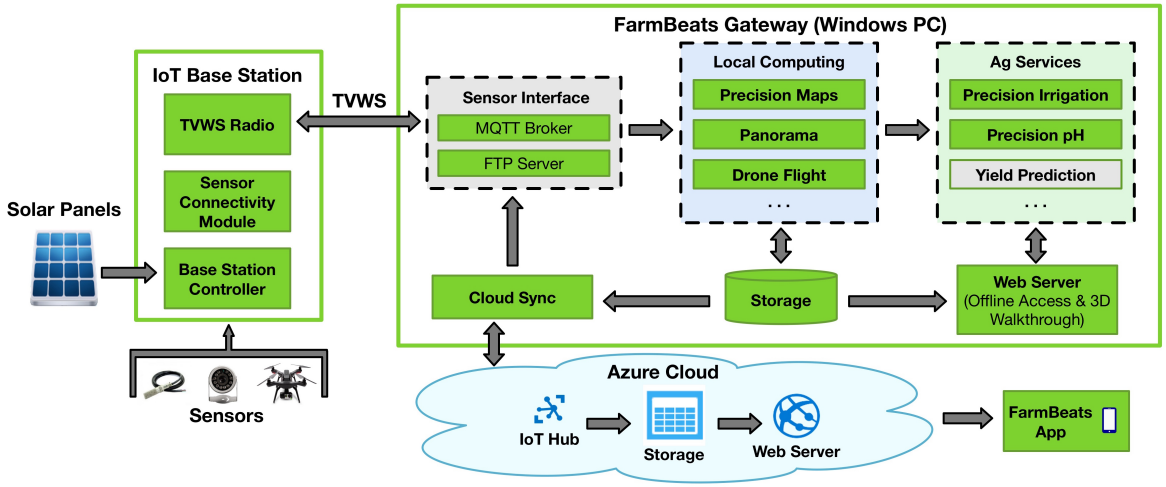


Figure 1: **FarmBeats System Overview**

as shown in Table 1), farms do not have access to power and high-bandwidth Internet connectivity unlike indoor IoT systems. Furthermore, energy harvested from the environment and weak network connectivity to the farm is susceptible to failures due to weather variability. So, the key question for the design of FarmBeats is: how does one design an IoT platform to meet the objectives in a highly variable, resource constrained environment?

3.1 Design Decisions

An overview of the system is given in Figure 1. Here, we discuss the main design decisions.

To achieve farm connectivity over long range, we leverage recent work in the TV White Spaces [6, 16, 44] to setup a high-bandwidth connection from the farmer's home to the farm. However, sensors, drones and cameras typically do not support TVWS. Thus, in order to maintain compatibility with sensors along with long-range high bandwidth connectivity, we deploy a two-layer hybrid network. We use a TVWS link to connect the farmer's home Internet connection to a few IoT base stations on the farm. Since it is a high bandwidth backhaul link, each base station can accommodate sensors, as well as cameras and drones. At the second layer, the IoT base station provides a Wi-Fi interface for connections from sensors and other devices. The Wi-Fi interface ensures that the farmer can not only connect most off-the-shelf farming sensors, cameras and drones; but they can also use their phone to access farming productivity apps. [2]

Variability in harvested solar energy leads to IoT base station downtime in overcast conditions. In fact, in our early deployments, power failures due to environmental factors were the major cause of unavailability. While past work has dealt with this problem in the context of single sensors [22, 51, 59] by duty cycling the sensors, the same approach does not work for a base station. Specifically, the base station has multiple components with different power requirements and duty cycling costs. For example, a farmer is typically inactive at night and is unlikely to check the farm data. So, turning the TVWS device off (which consumes 5x more power than the rest of the base station) can enable the base station to collect data (in a cache) from the sensors more frequently. Further, FarmBeats enables the farmer to turn the base station on to access Wi-Fi for productivity applications, while they are on the farm. This adds another layer of uncertainty in the duty cycling plan. Thus, we propose a novel duty cycle policy (in Section 4) wherein the different components of the base station are duty cycled at different rates; while explicitly accommodating these constraints.

Finally, given the weak internet connectivity to the farm, a naive approach of pushing all the data to the cloud does not work. We make the key observation that the data requirements of the farming applications can be broadly classified into two main categories: immediate detailed data and long-term summarized data. Table 2 summarizes how the industrial and research applications of farm data can be classified into these two categories. This categorization enables a gateway based IoT design for FarmBeats. The local gateway sits at the farmer's home at the other end of the White Space link and performs two functions: a) creates summaries for future use and ships them to the cloud and b) delivers applications that can be provided locally. The summaries are several orders of magnitude lower in size than the raw farm data (3-4 orders of magnitude smaller in case of the precision agriculture application discussed later) and hence, respect the harsh bandwidth constraints.

3.2 Architecture

The FarmBeats system has the following components: **Sensors & Drones:** FarmBeats uses off-the-shelf sensors for its applications. Each sensor measures specific

[2] Future iterations of the systems would add multiple interfaces to the base station to enable compatibility with more sensor types.

Technology	Cost	Data Restriction
Cellular Connection (Decagon Devices)	Per sensor fee: 1000$ + 100$ annual fee	Restricted to sensor data; Uploads every 15 mins at best
Mesh Networks (Ranch Systems)	Base station: 3500$ + 750$ annual fee; Per sensor fee: 1100$ + 60$ annual fee	Maximum 25 mesh nodes per base station
Satellite (Iridium)	Per sensor fee: 800$ + 100$ monthly fee	Restricted to 2.5 Kbps

Table 1: Cost Comparison of Farm Sensor Networking Solutions

Data Requirement	Applications
Immediate Descriptive Data	Precision irrigation, virtual walkthroughs, productivity apps, farm monitoring, ...
Long-term Summarized Data	Crop suggestions, seed distribution, yield monitoring, financial management, animal health statistics, ...

Table 2: Application classification based on requirements

characteristics of the farm, such as soil moisture and soil pH, and reports this data to the IoT base station over a Wi-Fi connection. In addition to soil sensors, FarmBeats supports cameras for farm monitoring and drones. The cameras are either connected to the IoT base station over Ethernet or report data over Wi-Fi. They take periodic snapshots and transmit this data to the IoT base station. UAV flights are either periodically scheduled or manually initiated using the FarmBeats app on the farmer's phone.

IoT Base Station: The IoT base station on the farm is powered by solar panels, backed by batteries and has three components:

- The TVWS device ensures that the base station on the farm can send the data to the gateway, which then, sends it up to the cloud.
- The sensor connectivity module establishes a connection between the base station and the sensors deployed on the farm. In FarmBeats's current implementation, this module is just a Wi-Fi router.
- Finally, the Base Station Controller is responsible for two functions. First, it serves as a cache for the sensor data collected by the sensor module and syncs this data with the IoT gateway when the TVWS device is switched on. Second, it plans and enforces the duty cycle rates depending on the current battery status and weather conditions.

IoT Gateway: As mentioned before, the goal of the IoT gateway is to enable local services and create summaries from existing data to be sent to the cloud. We use a PC form factor device as the FarmBeats gateway, which is typically placed in the farmer's house or office, whichever has Internet access. The gateway provides an interface for applications to run and create summaries to be sent to the cloud as well as to post data to the local web server. Furthermore, it includes a web service for the farmer to access detailed data when they are on the farm network. This also ensures that FarmBeats remains available even when the cloud connection is not present. Finally, it includes built-in algorithms for drone path planning and for compressing drone data before being sent to the cloud (described in Section 5). We illustrate in Section 5.3 how applications function on the gateway with the example of precision agriculture applications.

Three aspects of the FarmBeats gateway differentiate it from prior IoT gateways. First, the FarmBeats gateway implements a web service, providing unique services that are different from the FarmBeats web service in the cloud. Second, the gateway can operate offline, and still offer the most important services. Finally, as shown later in the context of precision agriculture, having access to data from multiple types of sensors enables unique feature-based summarization technologies for the drone videos and sensor data.

Services & the Cloud: The Gateway ships data summaries to the cloud, which provides a storage system for long-term data and a web interface for the farmer. The cloud enables three functions: data access outside the farm network (e.g. when traveling), long term applications like crop suggestions, and cross-farm analytics.

4 Duty Cycling the Base Station

As discussed before, FarmBeats's solar-powered IoT base station on the farm is duty cycled to explicitly account for weather forecasts and current charge state of the batteries. Two aspects of the base station make this problem challenging: a) The sensor connectivity module has significantly lower power requirements than the TVWS device. Thus, we need to intelligently proportion power between these components to achieve optimum performance. b) FarmBeats allows farmers to manually turn the base station on to connect to the Internet to use productivity apps on their phone. This adds a variable component to power consumption.

4.1 Duty Cycling Goals

The key goals for the duty cycling algorithm are:

- **Energy Neutrality:** Like past work in the context of duty-cycling sensors backed by energy harvesting sensor systems [22,59], FarmBeats aims to achieve the objective of energy neutrality. For a given planning period, the goal is to consume at max as much power as can be harvested from the solar panels.
- **Variable Access:** FarmBeats allows farmers to access Wi-Fi connectivity on-demand. This power consumption

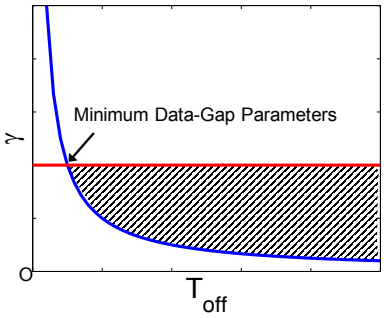


Figure 2: **Duty Cycling Approach:** The shaded region shows the feasibility region of Equations 2 and 4. The latency is minimized when both the equations are satisfied on the boundaries.

is usage-driven and varies across days. FarmBeats must plan ahead for this variable delay.

- **Minimize Data Gaps:** We use the term 'Data Gaps' to denote continuous time-intervals with no sensor measurements available. Such gaps need to be minimized to avoid missing out on interesting data trends. So, FarmBeats's duty cycling algorithm aims to minimize the length of the largest data gaps, under the constraints of energy neutrality and variable access.

4.2 Power Budget

The sole power source for the base station is a set of solar panels (backed by a battery). The solar power output varies with the time of day and the weather conditions. We use standard methods [51] to estimate the output of the solar panels, given the weather conditions. Let us say that the energy output from the solar panels over the next planning period is S_I. Because the estimation is not perfect and there is usage variability, there maybe some credit or debit from the previous planning period. Let us denote this credit by C_I. So, the total power budget for the base station over the next planning period is $S_I + C_I$.

4.3 Duty Cycling Approach

The duty cycle decisions are made on the order of a planning period, T_p. Since our deployments use solar powered base stations, we set T_p to be one day. We define the average energy loss due to battery leakage and the very low power base station controller during one T_p to be E_D. For the farmer to have on-demand Wi-Fi access, we allocate a fixed time budget of T_v. If we denote the power consumption of the TVWS device by P_T and the power consumption of the sensor connectivity module by P_S, then, we need to allocate $T_v(P_T + P_S)$ for variable Wi-Fi access. Now the key question is, how do we proportion the remaining power budget?

Duty Cycling the TVWS device: The TVWS module is needed to sync the data in the base station cache with the gateway. Let us assume that we have a schedule, S, the set of sync times advised for the base-station to sync with the FarmBeats gateway. This could depend on the farmer's usage patterns, sensor types and can be either manually programmed or automatically inferred. The sync times in the set S have a corresponding set of weights given by set W. An example of a high-weighted sync time could be sunrise, as that is when the farmer begins their day. Thus, they would like to access the latest sensor data when the activities of the day are planned.

To ascertain the subset of syncs that need to be performed, we make a simple observation. If the sensors haven't sent any data to the base station, the base station need not turn on the TVWS device. Specifically, it uses the following greedy algorithm to identify the syncs to be executed. Let us denote by, $S_1 \subset S$, the subset of syncs that are to be executed. This subset is initialized as an empty set. FarmBeats starts by adding the highest priority sync to S_1. After it has done that, it subtracts $|S_1|P_T T_S$ from the power budget, where $|.|$ denotes set cardinality and T_S denotes the time to perform a sync operation. Then, FarmBeats computes the corresponding duty-cycle rate for the sensor connectivity module. If this rate ensures that the second highest weighted sync in S will have additional data from the sensors to sync with the gateway, it adds this sync operation to the set S_1. It repeats this process in decreasing order of weights until it reaches a state where one of the syncs in S_1 has no new data to share. As we add more sync operations to S_1, the power budget for the sensor connectivity modules decreases. With a lower power budget, the sensor connectivity module can collect data from the sensors less often and hence it becomes less likely for frequent syncs to see new data. Thus, the algorithm implicitly regulates the sync operations between the gateway and the base station.

Duty Cycling the Sensor Connectivity Module: We denote the duty cycling rate for the sensor connectivity module by γ. In particular, it is turned off for a time period, T_{off}, followed by an on period of, T_{on} and $\gamma = T_{on}/T_{off}$. Using the notation we have established so far, the energy expenditure of the system is $E_D + (P_S + P_T)T_v + P_T T_S|S_1| + P_S T_p \gamma$. Since the goal of the planning algorithm is to estimate T_{on} and T_{off} such that the energy expenditure does not exceed the energy budget during the planning period, this imposes the following constraint:

$$S_I + C_I \geq E_D + (P_S + P_T)T_v + P_T T_S|S_1| + P_S T_p \gamma \quad (1)$$

$$\implies \gamma \leq \frac{S_I + C_I - E_D - (P_S + P_T)T_v - P_T T_S|S_1|}{P_S T_p} \quad (2)$$

Let us denote $T_{connect}$ as the time taken for the sensor connectivity module to turn on and establish a connection to the sensors. Further, let T_{sensor} be the time that it takes for all the sensors to wake up and transmit to the base station. Since the ON time of the module has to be long enough for the sensors to be able to communicate their data to the base station, this imposes a further constraint:

$$T_{ON} \geq T_{connect} + T_{transfer} \quad (3)$$

$$\implies \gamma T_{off} \geq T_{connect} + T_{transfer} \quad (4)$$

Since our goal is to minimize the data gap under the power constraints imposed by Equations 2 and 4, we aim to minimize T_{off}. The inequalities from Equations 2 and 4 define a convex region in the 2-dimensional space of (γ, T_{off}), shown as the shaded region in Figure 2. Since the cost function T_{off} is linear, the minimum occurs on a corner of the intersection region defined by the two inequalities. Specifically, the minimum latency is achieved when the two inequalities are exactly met. The solution is shown graphically in Figure 2.

4.4 Discussion

At this point, it is worth noting that:

- By explicitly accounting for the credit term, C_I, the formulation absorbs the variability in on-demand Wi-Fi usage patterns. If the on-demand Wi-Fi usage patterns are stable, the term C_I goes down to zero.
- By incorporating flexibility in sync times between the gateway and the base station, FarmBeats can easily adapt to farm applications with different requirements.
- We have not yet discussed the duty cycling of sensor nodes. In our implementation, we set the duty cycle off time for sensors to be less than $T_{transfer}$ to ensure that the sensor can transfer data when the sensor connectivity module is on. An alternative implementation would allow the base station to send wake-up times to sensors. Our design choice was motivated by the availability of very low-power sensors that consume 3-4 orders of magnitude less power than the base station on average.

5 THE FARMBEATS GATEWAY

In this section, we discuss two key components of the FarmBeats gateway: UAV path planning and stitched imagery (orthomosaic) generation from UAV videos. We also illustrate how the FarmBeats gateway implements a precision agriculture application.

5.1 UAV Path Planning

Most UAVs operate in line sweep patterns. Specifically, given a sequence of waypoints defined by their GPS coordinates, they move from one waypoint to the next, in order. However, in the context of agriculture, our objective is to optimize for the area covered in a single flight. Thus, we aim to minimize the time taken to cover a given area. To that end, we make the observation that increasing the number of waypoints to cover the same area increases the time taken to cover it, even though the total path length may be the same. This is because the quadrotor has to decelerate at each waypoint and come to a halt before it can turn around and accelerate again. We present a novel flight planning algorithm that minimizes the number of waypoints required to cover a given part of the farm.

Existing commercial systems like Pix4D [42], DroneDeploy [15], etc. offer area coverage services, these systems cover a given area using an east-to-west flight path, without any regards to the number of waypoints required. Recent research proposals like [17] do not guarantee the minimum number of waypoints either. Our area coverage algorithm *Min-waypoint* described below guarantees that the UAV covers an area with the minimum number of waypoints:

- Given an area, construct its convex hull.
- Determine the direction of sweeping lines. For each edge and its antipodal (diametrically opposite) vertex [50], draw two parallel lines and measures the distance between them. The slope of the edge corresponding to the minimum distance between the edge and the antipodal vertex becomes the direction of the sweeping lines.
- Determine the waypoints depending on the flight altitude, the camera's field of view, and desired image quality.
- Given a start-point and end-point of the flight path, order the waypoints to minimize the total travel distance.

Adaptive Wind-assisted Yaw Control: Since farms are large open spaces and typically very windy, we observed that quadrotors that have an asymmetric physical profile can exploit the wind either for more efficient propulsion or deceleration. Figure 3(b) shows an example of a quadrotor (DJI Inspire 1) that has an asymmetrical profile, where its front and the side are considerably different; thus, it can exploit the wind similar to sailboats. Intuitively, when the quadrotor is flying downwind (i.e. wind is helping the quadrotor), the side profile of the quadrotor should face the wind since the side profile has a larger area and hence, will be able to extract the maximum assistance from the wind. In our experiments on the farm, the quadrotor requires significantly more energy (80% higher at 4m/s) to maintain its speed upwind in comparison to the downwind flight.

To leverage this observation, we designed a novel yaw control algorithm to exploit the wind energy on the farm. Specifically, yaw is the angle of the quadrotor with respect to the vertical axis. While we don't describe the algorithm in detail, on a high level, Figure 3 describes how the yaw control algorithm would operate for a quadrotor that has a larger area on the sideways profile. For the downwind segment from the start point to the first waypoint, the adaptive control starts by making the yaw perpendicular to the flight path, thereby maximally utilizing the favorable wind as the quadrotor accelerates. However, as the velocity increases, the air drag generated by the quadrotors profile also increases. Consequently, once the quadrotor accelerates the yaw is reduced so as to maximally exploit the wind, while minimizing the parasitic drag due to the side profile. Similarly, the deceleration

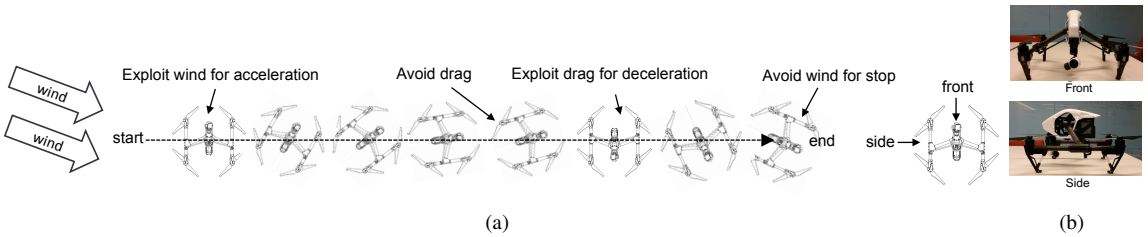


Figure 3: FarmBeats's path planning algorithm uses the asymmetry in front and side profiles of a drone like DJI Inspire 1 (in (b)) to leverage wind to its advantage

phase can very effectively exploit the air drag by making its yaw perpendicular to the flight path. This action is analogous to the action that a skier takes to stop.

5.2 Generating Orthomosaics from UAV Videos

UAVs generate a prohibitive amount of video that is difficult to transfer to the cloud due to poor network connectivity on farms. For example, a 4 minute flight with a UAV capturing 1080p video at 30 frames per second generates almost a Gigabyte of video data. We make the observation that the unit of interest for the farmer is not the drone video itself, but an overview of the farm that can be provided by a geo-referenced panoramic overview, which is one-two order of magnitude more compact than the full resolution video (see Figure 4). The stitched orthomosaic generated from the drone video provides a high resolution visual summary of the farm from a low altitude vantage point, revealing minute details. In fact, existing agricultural drone solutions ([10, 49]) ship the videos to the cloud and convert them into orthomosaics to show to the farmer. Thus, we incorporate the orthomosaics processing pipeline into the FarmBeats Gateway, to process the drone videos locally.

Broadly speaking, the panoramic views can be constructed from the UAV video using two approaches, based on either (i) aerial 3D mapping [42, 48] or (ii) image stitching and mosaicking [4, 7, 36, 53, 56]. While the aerial 3D mapping is a general-purpose method to reconstruct high resolution 3D surface maps of the environment from aerial videos, the image stitching methods treat the world as planar and simply stitch the different images together by finding their relative positions.

Computing high-resolution surface maps is both compute and memory intensive and is not suitable for the resource-constrained farm gateway. On the other hand, while image stitching methods can be incorporated into the gateway, the planar terrain assumption becomes invalid on the farm. Uneven ground geometry, trees, animals or man-made structures observed in the video generates parallax which cannot be handled by the image registration algorithms that assume a planar scene. As we show later in Section 7 and as observed in prior work [27], existing image stitchers – Microsoft ICE [36], AutoPano [4] tend to produce distorted orthomosaics in such scenarios. This presents us with an uncomfortable tradeoff: either fly high such that the farm appears planar and sacrifice fine details of the farm, or ship the large aerial videos to the cloud for processing.

Our approach: In order to break this tradeoff, we have developed a hybrid technique which combines key components from both 3D mapping and image stitching methods. On a high level, we use techniques from the aerial 3D mapping systems, just to estimate the relative position of different video frames; without computing the expensive high resolution digital surface maps. Since this process can be performed at a much lower resolution, this allows us to get rid of the harsh compute and memory requirements, while removing the inaccuracies due to non-planar nature of the farm. Once these relative positions have been computed, we can then use standard stitching software (like Microsoft ICE) to stitch together these images. The performance achievements of this hybrid approach are evaluated further in Section 7.

5.3 Generating Precision Maps

As discussed before, precision agriculture relies on accurate precision maps of the farm that indicate the distribution of a specific characteristic throughout the farm. The FarmBeats gateway naturally enables a novel approach to precision map generation that can use the aerial imagery from drones to perform spatial inference of sensor values from sparsely deployed sensors.

Specifically, FarmBeats uses the orthomosaic generated from the drone videos together with the sensor values observed by the sensors planted in the soil, and generates predictions for the entire farm. For example, sensors that observed soil temperature at the discrete locations can inform the machine learning pipeline to make predictions about every location in the farm by considering spatial proximity as well visual similarity of the locations to the sites with the sensors.

FarmBeats's gateway embeds a machine learning pipeline that draws on probabilistic graphical models that embed Gaussian processes [43]. The key intuition in the proposed model is *spatial and visual smoothness*: areas that are similar should observe similar sensor readings. Specifically, the model relies on two kinds of similarities:

- **Visual Smoothness:** Areas that look similar have similar sensor values. For example, a recently irrigated area

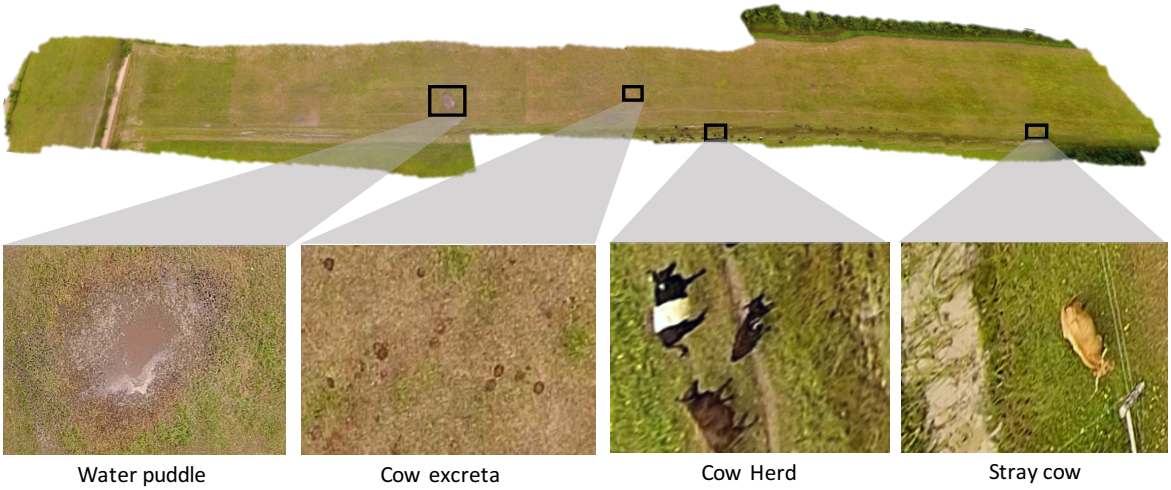


Figure 4: **Orthomosaic Generation:** The high resolution orthomosaic generated by FarmBeats for a 5 acre patch in the large farm reveals important visual details to the farmer, such as those shown in the insets – puddles that can make part of the land unavailable for agriculture, cow excreta that becomes manure and enriches the soil, location of individual cows grazing on the farm and their distance from the nearby electric fence.

would look darker and hence, has more moisture.

- **Spatial Smoothness:** Since we are measuring physical properties of the soil and the environment, the sensor readings for locations that are nearby should be similar.

We encode these two intuitions into a graphical model using standard techniques and formulate it as a Gaussian process regression model [43].

In our current design, FarmBeats uses the precision maps as units of summarization for the UAV data and ships them to the cloud. This has two advantages over the using orthomosaics as the unit of summary. First, they incorporate sensor data from the farm into drone videos. Second, they can be compressed to two to three orders of magnitude smaller size than a orthomosaic. So, while the orthomosaic is good for giving the farmer a detailed overview of the farm, precision maps are better for long term storage and shipping. We envision that for other machine learning applications as well, feature maps like the precision maps of the field would be the summaries that get shipped to the cloud, while the descriptive data delivers short-term applications on the gateway.

6 Deployment

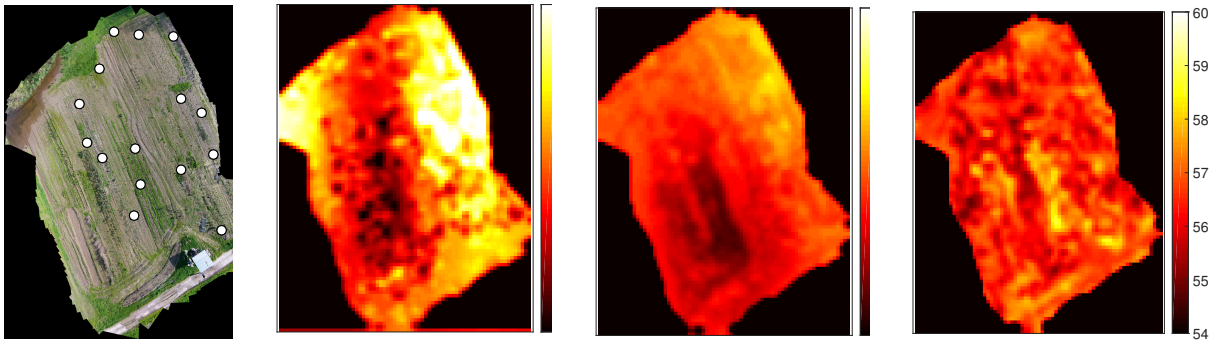
We deployed FarmBeats in two farms located in Washington (WA) state and in upstate New York (NY), with an area of 5 acres and 100 acres, respectively. The farmer in WA grows vegetables that he sells in the local farmers market. The farm in upstate NY follows the community supported agriculture (CSA) model, and grows vegetables, fruits, grains, as well as dairy, poultry, and meat. Our deployments consist of: sensors, cameras, UAV, the IoT base station, a gateway PC, the cloud service and a dashboard (mobile app and a web page).

Sensors: Each farm was equipped with sensors that measure soil temperature, pH, and moisture. In case of sensors without Wi-Fi support, we interfaced them with Arduinos, Particle Photons or NodeMCUs to add Wi-Fi capability. While the exact number of sensors varied over the deployments and the application of interest, we have deployed over 100 different sensors. Additionally, We deployed Microseven IP [33] cameras in different parts of the field to monitor the farm, as well as to capture IR images of crops. To avoid potential damage from environmental impacts, each sensing platform was encased in a weatherproof box. An example of a sensor deployment can be seen in Figure 6(a).

Drones: We used the DJI Phantom 2, Phantom 3 and Inspire 1 for our drone flights.[3] We created an auto-pilot application using the DJI Mobile SDK [12] to interface with FarmBeats. The user can use the app to first select the flight altitude and determine the area to be covered on an interactive map. FarmBeats's app then plans a flight path using the algorithm proposed in Section 5.1. After the drone completes its mission, it automatically returns to its home position and transfers the video recording during the flight to the gateway, through the IoT base station.

IoT Base Station: At each IoT base station deployment, we set up a TVWS network using the FCC certified Adaptrum ACRS 2 radios [2] operating at 20 dBm, and 11 dBi directional antennas with 90 degree sectors. The internet connectivity was provided by the home internet connection of the farmers. To power the base station we setup a solar charging system, which comprised of two 60 Watt solar panels connected to a solar charge controller. The powering system is backed by four 12V-44Ah batteries connected in parallel. The power output goes through an 8-port Digital Logger PoE switch [28]. This provides us the capability to turn on or off individual components of the base station. A Raspberry Pi 3 with 64 GB SD

[3] We received an exemption from the FAA to fly the UAV.



(a) Orthomosaic (b) Moisture Map (c) pH Map (d) Temperature Map

Figure 5: **Precision Maps:** (a) A 40 MPixel orthomosaic created from a 3 minute flight over 2 acre area of a farm. Our system infers dense sensor measurements from very few sensors deployed on the farm (indicated by white circles). (b) The predicted soil moisture map (our sensors measures moisture on a scale of 1 to 5). Note that the top left region in the image where the ground appears wet was correctly predicted to have high moisture even though no moisture sensors were present in that part of the farm. (c) The predicted pH map (pH is measured from 0-14, 7 is neutral and 0 is the most acidic). Our system identified that the whole field is slightly acidic, but the bottom left/center is more acidic than the rest. (d) The predicted soil temperature map (in Fahrenheit scale).

card serves as the base station controller. The sensors interfaced with the base station through a 802.11b router, with a range of over 100 m.

Gateway: The gateway is a Lenovo Thinkpad in the WA farm and a Dell Inspiron laptop in the upstate NY farm.

Cloud: We use the Azure IoT Suite ([34]) for FarmBeats. The sensor readings, camera images, and drone video summaries are populated through the Azure IoT Hub ([35]), to storage. We use blobs for images, and tables for the sensor readings. Although in our current implementation, the different farms share the Azure account, with table-level access control, we plan to have different cloud service accounts for the different farms, as FarmBeats scales up.

7 RESULTS

We evaluate the components of FarmBeats below:

7.1 Weather Aware Base Station

The FarmBeats base station leverages the algorithm in Section 4 to duty cycle different components. It uses the OpenWeather API [40] to get the weather forecasts and plans the duty cycling scheme for the next day. The weather information gives us the cloudiness percentage for each period of three hours. The cloudiness percentage over three days is plotted in Figure 7(a).

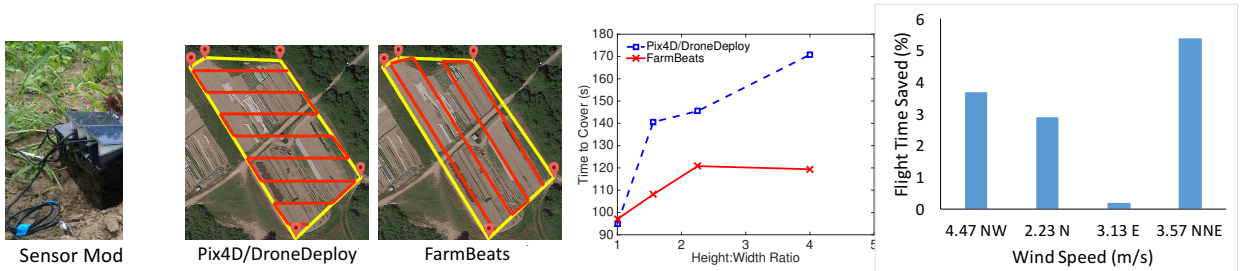
Over this set of three days, we compare three power-awareness schemes. We define the start of the day as 6AM local time. We periodically record the state-of-charge of our solar power backed batteries. First, we let the base station be always on. As shown in Figure 7(b), the battery charge goes up during a sunny day and down during the night. While the base station remains energy neutral during the first day, during subsequent days its battery drains because of cloudy weather, leading to unavailability on the third day. Then, we evaluate the alternate approach. We set the base station to a conservative duty cycling period. While this ensures that the base station is available on cloudy days, the base station battery charges up to 100% during the sunny days thus wasting solar power that could have been utilized. Moreover, its duty cycling interval collects *15 times* less data than the optimal FarmBeats solution, plotted in 7(d).

FarmBeats collects data on the first two days more frequently owing to high availability of solar power. However, on the third day, it switches to a conservative duty cycling schedule to save power. Of the 15x gain in data collection frequency achieved over a fixed duty cycle, a factor of 2 is because of the TVWS client being duty cycled at a different rate than the Wi-Fi router. An earlier version of our deployment which did not duty cycle the base station faced a downtime of 30% in a cloudy month as opposed to zero downtime for our power-aware design in the same month. Thus, FarmBeats's power-aware design achieves its goal of maximizing data-freshness while maintaining energy neutrality.

7.2 UAV Flight Planning

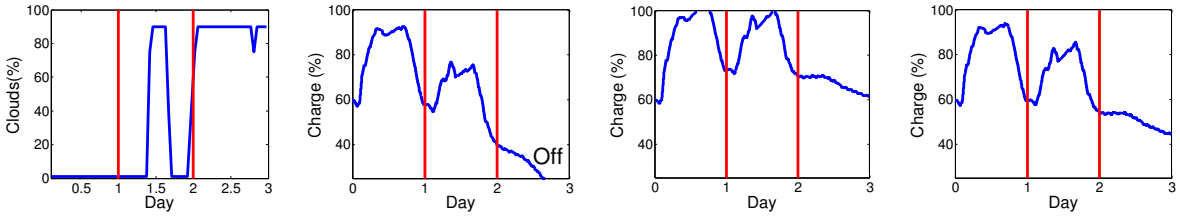
As mentioned in Section 5.1, we use an efficient area coverage algorithm in addition to leveraging wind assistance to extend drone battery life. To understand the impact of area coverage algorithms on drone flight time, we compare performance of FarmBeats in covering a given area as compared to the state-of-the-art *East-to-west* algorithm (used by Pix4D, DroneMapper, etc.). As shown in Figure 6(b), the east-to-west algorithm generates sweeping patterns from the east to the west or vice-versa regardless of the area shape. However, FarmBeats generates a path that minimizes the number of waypoints.

Next, we compare the time taken to complete flights planned by the two algorithms to cover a given area. The maximum speed was set to 10m/s and the altitude was set to 20m. Figure 6(c) plots the time taken to complete a flight with the two algorithms in different area geome-



(a) (b) (c) (d)

Figure 6: (a) A weather-resistant, solar-powered FarmBeats sensor module.(b,c,d) Drone Flight Planning: (b) FarmBeats's flight planning algorithm minimizes the number of waypoints to cover a region. (c) Depending on the aspect ratio of the field, flights without FarmBeats's algorithm take upto 42% more time. This improves the time by a factor of 1.26 in the average case for our farms. (d) In addition, the yaw control algorithm described in Section 5.1 achieves a gain of up to 5% based on the wind velocity.



(a) Cloud Cover (b) No Duty Cycle (c) Fixed Duty Cycle (d) FarmBeats

Figure 7: **Power-aware Base Station:** The cloudiness percentage over 3 days. (b) With no duty-cycling, the base station shuts down on a cloudy day. (b) A fixed conservative duty cycle can prevent the base station from going down, but it collects 15 times less sensor data. (d) FarmBeats's Power-aware basestation can keep the base station on by reducing the duty-cycling on days are expected to be cloudy.

tries defined by their height to width ratio, where height is the distance along the North-South direction and width is measured along East-West. As expected, the gain achieved by FarmBeats increases as the height-width ratio increases. This is because FarmBeats algorithm generates fewer waypoints to cover the same area. In general, for the average case of our deployments, FarmBeats reduced the time taken to cover an area by 26%.

Finally, we evaluate the impact of our yaw control algorithm under different wind conditions. The maximum speed was set to 10m/s and the altitude was set to 30m. For every flight, we fully charged the battery. We measure the percentage of time saved by FarmBeats's yaw control algorithm for each flight and plot it in Figure 6(d). As seen in the figure, FarmBeats can save up to 5% time depending on the wind velocity. Moreover, as the north-south component (the principal direction of motion for this set of experiments) of the wind increases, FarmBeats can leverage it better.

7.3 Orthomosaic Generation

The novel orthomosaic generation algorithm proposed in this paper advances the state-of-the-art on two fronts. First, our approach of combining sparse 3D reconstruction techniques from video with image stitching techniques is more robust than existing techniques based on either aerial 3D mapping or aerial image stitching. In addition, our approach is computationally more efficient and runs considerably faster than Pix4D [42], an aerial 3D mapping-based tool catering to Precision Agriculture.

Qualitative Results: We show two representative orthomosaics constructed by FarmBeats and Microsoft ICE in Figure 4 and 8(b) respectively. Figure 8(a) shows what the farm looked like in Google Earth in the past. The orthomosaic generated by Microsoft ICE failed in this case, while our result is consistent and accurate. Our geo-referenced image covers about 5 acres of farmland and provides a detailed visual summary to the farmer. By visually inspecting the high-resolution image, they can discover anomalies such as the water puddle that can render a part of the field unsuitable for agriculture for a couple of seasons. Moreover, the farmer can see where cows are grazing during the day and make a decision about whether they want to move them to another spot for the next day. The decision is based on how much grass they want to leave on the field to be converted into manure.

Processing Time: As shown in Figure 8(c), our implementation is 2.2 times faster than Pix4D on average. Specifically, our method took 14 minutes to construct an orthomosaic on average whereas Pix4D took 32 minutes on average on a set of videos captured by our drones at 1080p resolution at 30 frames per second. This demonstrates the improved running time of our method.

Finally, the orthomosaic generated by our system are approximately 5 times smaller than the original video size at full resolution (in .png format) before applying lossy compression. A single pixel in the geo-referenced orthomosaic is about 2 *cm* in size which is equivalent to a single penny on the ground. The image resolution and compression quality are parameters that can be tuned to meet any target file size.

(a) (b) (c)

Figure 8: **Orthomosaic Generation:** (a) The Google Earth image for the farm in Figure 4. (b) Microsoft ICE image stitching pipeline fails to reconstruct it accurately. (c) Pix4D takes about 2.2x longer on average compared to our approach.

7.4 Generating Precision Maps

As described in Section 5.3, FarmBeats uses the visual features from the orthomosaic overview to extrapolate the sensor values and generate precision maps for soil temperature, soil moisture and pH.

Qualitative Evaluation: We show a representative set of these precision maps in Figure 5. As shown in the figure, based on sensor values in the rest of the farm, the moisture prediction pipeline can estimate that the top left part of the farm has high moisture content even though that part has no sensor there. Similarly, the pH map generates an actionable input in the sense that the bottom left and center of the farm have very low pH and are highly acidic. As a result of this map, the farmer applied lime to enhance the pH and make the soil more neutral.

Note that the pH of the farm varies within the farm at fine granularity. As seen in Figure 5, within a couple of acres, the pH can vary from 4 (very acidic) to 7 (neutral). Soil moisture variance is even higher, with variance seen within a few meters. Precision maps generated by FarmBeats capture this variance accurately, by using the drone videos to extrapolate the sensor data.

Quantitative Evaluation: In order to evaluate the accuracy of the precision maps generated by FarmBeats using the approach described in Section 5.3, we evaluated our system on 5 datasets constructed from the drone videos and sensor data. Each dataset corresponds to a drone flight over the farm (covering 2 acres) and one set of sensor measurements from the sparse sensor deployment. The hyperparameters are learned by doing 5 fold cross validation. As an accuracy metric, we measure the correlation between the predicted sensor values and the ground truth sensor values to see how well the variations in the field are captured by FarmBeats. We compare against two techniques, which do not use the drone video based extrapolation of the sensor values:

- **Nearest Neighbor (SensorsNN):** We assign the value from the nearest sensor to each point in the field.
- **Inverse Distance based Interpolation (SensorsInterp):** We linearly interpolate known sensor values in the field, by using inverse distance as a weight. This technique has been previously been proposed in the context of precision agriculture [14, 55].

For all the analysis, we use leave-one-out evaluation, i.e., we generate a precision map after leaving one of the sensors out of the training set and evaluate the map on the left out sensor. We repeat this process for all the ground sensors and report the averaged results.

The comparison of correlation across the different schemes is shown in Figure 9(a). As shown in the figure, FarmBeats outperforms existing sensor based interpolation techniques. In particularly, FarmBeats can accurately estimate the variations of the different sensor values in the field. While sensor based methods do not mirror the variations and hence have nearly zero correlation with the sensor values, FarmBeats's estimates have high positive correlation with the true sensor values, thus indicating the utility of using the drone video in conjunction with the drone estimates. Finally, the precision maps generated by FarmBeats are 3 orders of magnitude smaller in size on average than the video and can be easily shipped to the cloud during periods of connectivity.

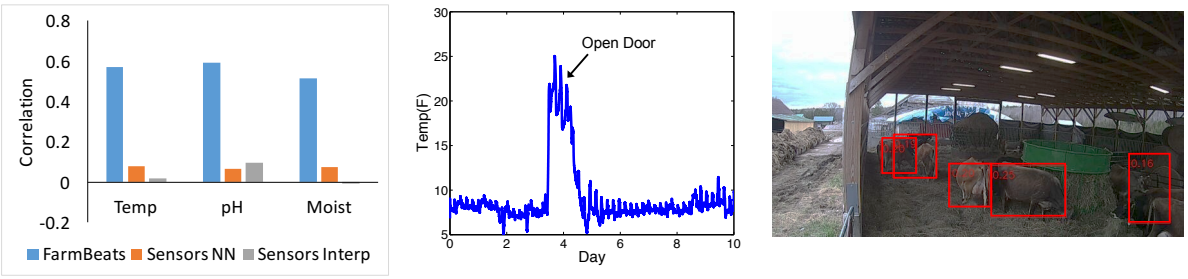
7.5 Other Applications

Figure 9 highlights two other applications that the farmers used FarmBeats for. First, the farmer in NY used FarmBeats sensors to monitor his storage freezers. The temperature in these freezers is carefully regulated below 10° F to prevent produce from going bad. As shown in Figure 9(b), an employee leaving the door open could lead to this temperature going up causing loss to the farmer. This problem is solved by FarmBeats by enabling automated notifications based on these sensor readings in the FarmBeats phone application.

Second, the farmers plugged in cameras at different locations like cow sheds and connected them to the nearest FarmBeats base station. One frame of the camera is shown in Figure 9(c). While the intent of the current application is to manually monitor the cows, one can potentially build an application that can detect anomalies in cow behavior or use cow motion to track animal health [38]. As a preliminary result, we ran a deep neural network based cow detector on the data. The identification boxes are overlaid on the figure.

7.6 End-to-end Deployment Statistics

Data Aggregation: FarmBeats's deployments at both farms have been running for over six months. Over these deployments, FarmBeats interfaced with around 10 different sensor types, three different camera types, three versions of drones and the farmers' phones. It collected



(a) Precision Maps (b) Storage Monitoring (c) Cow Shed Monitoring

Figure 9: **FarmBeats Applications** (a) FarmBeats's precision maps are more accurate than standard sensor based interpolation techniques. (b) Temperature (measured in F) in a storage unit can raise an alarm when an employee leaves a door open. (c) Cows being monitored in a cow shed. The red boxes indicate a standard cow detector output.

more than 10 million sensor measurements, half million images and 100 drone surveys.

Resilience to Outages: FarmBeats's deployments faced one week-long Internet outage due to a thunderstorm and several smaller term Internet outages. The FarmBeats gateway continued to be available during these times.

Cost: The TVWS client radios cost $200,[4] and there are no additional data charges, than the farmer's existing internet connection. The Particle Photons cost about $20 and can add Wi-Fi support to each sensor. Thus, use of the hybrid networking approach reduces the system cost by an order of magnitude as compared to existing systems which cost over $1000 in equipment cost per sensor and over 100$ annual subscription fee (see Table 1).

Applications: Farmers used FarmBeats's precision agriculture system to guide their precision irrigation units. The precision pH maps generated were used by farmers to apply lime in the more acidic regions. As mentioned before, farmers also used FarmBeats for storage monitoring with sensors and animal shelter monitoring, selling station monitoring with cameras. Beyond that, farmers also used FarmBeats base stations to access Wi-Fi while on the farm to run productivity applications like Trello.

8 Related Work

FarmBeats builds on past work in wireless sensor networks, precision agriculture and ICTD.

Wireless Sensor Networks: Past work has used multi-hop networks [5, 19, 23, 26, 39, 45, 57, 60] to gather data from sensors in the farm. However, all these systems suffer from bandwidth constraints that make them unable to support sensors, cameras and drones. Further, these systems do not account for constraints imposed by weak cloud connectivity and weather related power and Internet outages. The same is true for recent advances in LPWAN technologies [29, 52]. In contrast, FarmBeats includes support for sensors, cameras and drones; is backed by cloud connectivity and has mechanisms to adapt to weather variability.

Agriculture: Agronomists have studied various aspects of precision agriculture, from defining more accurate management zones [31], to improving prescription [37], to leveraging soil science [54] and plant physiology [8] techniques. Prior work has also looked at applications of precision agriculture to irrigation, variable seeding, nutrient application, and others. There has been prior work on developing technology for enabling precision agriculture. Researchers have built specialized sensors for measuring nutrients [25], water levels [21], and other such sensors, and we build on top of this work. FarmBeats's work is complementary to this body of work as it facilitates the automation of data collection using these sensors and enables the precision agriculture systems.

ICTD: ICTD solutions focus on user interfaces to make existing technologies more accessible [13] enhanced access to information [9] and better communications. The mechanisms of data collection is manual in most scenarios. The few attempts at automated data collection, like [9], fall into the same pitfalls as discussed before. We believe FarmBeats is complimentary to this work and will aid the proliferation of ICTD by enabling end-to-end IoT connectivity in weakly connected scenarios.

9 Conclusions & Future Work

FarmBeats is a low-cost, highly available IoT platform for agriculture. It supports high bandwidth sensors using TVWS, which is a low-cost, long range technology. FarmBeats uses a weather-aware solar-powered IoT base station, and an intelligent Gateway that ensure that services are available in the Cloud and offline. It also incorporates new path-planning algorithms that extend drone battery life. We have deployed the system in two farms, and the farmers are already using it for three applications: precision agriculture, animal monitoring, and storage monitoring. Moving forward, we are working with the farmers to develop several other applications on top of FarmBeats. Further, we plan to make anonymized data available for researchers to enable more agricultural applications.

[4] With the standardization of IEEE 802.11af [1] standard, we expect the price to the client and base station to be similar to Wi-Fi, of less than 10$. We are testing one such multi-mode TVWS/Wi-Fi chip from a major Wi-Fi vendor.

REFERENCES

[1] IEEE 802.11af: https://standards.ieee.org/findstds/standard/802.11af-2013.html.

[2] Adaptrum. http://www.adaptrum.com/.

[3] M. H. Almarshadi and S. M. Ismail. Effects of Precision Irrigation on Productivity and Water Use Efficiency of Alfalfa under Different Irrigation Methods in Arid Climates. *Journal of Applied Sciences Research*, 2011.

[4] AutoPano. kolor.com.

[5] A. Baggio. Wireless sensor networks in precision agriculture. *ACM Workshop on Real-World Wireless Sensor Networks*, 2005.

[6] P. Bahl, R. Chandra, T. Moscibroda, R. Murty, and M. Welsh. White Space Networking with Wi-Fi like Connectivity. *ACM SIGCOMM Computer Communication Review*, 2009.

[7] M. Brown and D. G. Lowe. Automatic Panoramic Image Stitching Using Invariant Features. *International Journal of Computer Vision*, 2007.

[8] K. G. Cassman. Ecological Intensification of Cereal Production Systems: Yield Potential, Soil Quality, and Precision Agriculture. *Proceedings of the National Academy of Sciences (PNAS)*, 1999.

[9] J. chun Zhao, J. Zhang, Y. Feng, and J. Guo. The Study and Application of the IOT Technology in Agriculture. *IEEE International Conference on Computer Science and Information Technology*, 2010.

[10] DataMapper. http://www.precisionhawk.com/datamapperinflight.

[11] Decagon Devices. Decagon Devices Cellular Logger. https://www.decagon.com/en/data-loggers-main/data-loggers/em50g-wireless-cellular-data-logger/.

[12] DJI. http://developer.dji.com.

[13] J. Doerflinger and T. Gross. Sustainable ICT in Agricultural Value Chains. *IT Professional*, 2012.

[14] T. A. Doerge. In *International Plant Nutritional Institute*, 1999.

[15] DroneDeploy. dronedeploy.com.

[16] Federal Communications Commission. https://www.fcc.gov/general/white-space-database-administration.

[17] C. D. Franco and G. Buttazzo. Energy-aware coverage path planning of uavs. In *International Conference on Autonomous Robot Systems and Competitions (ICARSC),* , 2015.

[18] FreeWave. http://www.freewave.com/.

[19] A.-J. Garcia-Sanchez, F. Garcia-Sanchez, and J. Garcia-Haro. Wireless Sensor Network Deployment for Integrating Video-surveillance and Data-monitoring in Precision Agriculture over Distributed Crops. *Computers and Electronics in Agriculture*, 2011.

[20] H. C. J. Godfray, J. R. Beddington, I. R. Crute, L. Haddad, D. Lawrence, J. F. Muir, J. Pretty, S. Robinson, S. M. Thomas, and C. Toulmin. Food Security: The Challenge of Feeding 9 Billion People. *Science*, 2010.

[21] B. R. Hanson and S. Orloff. Monitoring soil moisture for irrigation water management. Technical report, UC Davis, 2007.

[22] J. Hsu, S. Zahedi, A. Kansal, M. Srivastava, and V. Raghunathan. Adaptive Duty Cycling for Energy Harvesting Systems. In *International Symposium on Low Power Electronics and Design*, 2006.

[23] O. V. K. Langendoen, A. Baggio. Murphy loves Potatoes: Experience from a Pilot Sensor Network Deployment in Precision Agriculture. In *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2006.

[24] Kathleen McLaughlin . Gaps in 4G Network Hinder High-tech Agriculture: FCC Prepares to Release 500 Million to Improve Coverage.

[25] H.-J. Kim, K. A. Sudduth, and J. W. Hummel. Soil Micronutrient Sensing for Precision Agriculture. *Journal of Environmental Monitoring, The Royal Society of Chemistry*, 2009.

[26] W. Lee, V. Alchanatis, C. Yang, M. Hirafuji, D. Moshou, and C. Li. Sensing Technologies for Precision Specialty Crop Production. *Computers and Electronics in Agriculture*, 2010.

[27] Z. Li and V. Isler. Large Scale Image Mosaic Construction for Agricultural Applications. *IEEE Robotics and Automation Letters*, 2016.

[28] D. Logger. PoE Switch. http://www.digital-loggers.com/poe48.html.

[29] LoRa Technology. https://www.lora-alliance.org/what-is-lora/technology.

[30] J. Lowenberg-DeBoer. The Precision Agriculture Revolution: Making the Modern Farmer. https://www.foreignaffairs.com/articles/united-states/2015-04-20/precision-agriculture-revolution.

[31] A. McBratney and M. Pringle. Estimating average and proportional variograms of soil properties and their potential use in precision agriculture. *Precision Agriculture*, 1999.

[32] A. McBratney, B. Whalen, T. Ancev, and J. Bouma. Future Directions of Precision Agriculture. *Precision Agriculture*, 2005.

[33] Microseven. Ip cameras. http://www.microseven.com/product/IP-Cameras.html.

[34] Microsoft. Azure IoT Hub. https://azure.microsoft.com/en-us/services/iot-hub/.

[35] Microsoft. Azure IoT Suite. https://www.microsoft.com/en-us/cloud-platform/internet-of-things-azure-iot-suite.

[36] Microsoft Research. Image Composite Editor. http://research.microsoft.com/en-us/um/redmond/projects/ice/.

[37] N. D. Mueller, J. S. Gerber, M. Johnston, D. K. Ray, and J. A. F. Navin Ramankutty. Closing Yield Gaps through Nutrient and Water management. *Nature*, 2012.

[38] L. Nagl, R. Schmitz, S. Warren, T. S. Hildreth, H. Erickson, and D. Andresen. Wearable Sensor System for Wireless State-of-health Determination in Cattle. In *Engineering in Medicine and Biology Society*, 2003.

[39] T. Ojha, S. Misra, and N. S. Raghuwanshi. Wireless sensor networks for agriculture. *Computers and Electronics in Agriculture*, 2015.

[40] OpenWeatherMap, Inc. OpenWeather API. http://openweathermap.org/.

[41] B. Ortiz, J. Shaw, J. fulton, and A. Winstead. Management Zones II – Basic Steps for Delineation. *Alabama Precision Ag Extension: Precision Agriculture Series*, 2011.

[42] PIX4D. Pix4d. https://pix4d.com/.

[43] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[44] S. Roberts, P. Garnett, and R. Chandra. Connecting Africa Using the TV White Spaces: From Research to Real World Deployments. In *IEEE LANMAN*, 2015.

[45] L. Ruiz-Garcia, L. Lunadei, P. Barreiro, and J. I. Robla. A Review of Wireless Sensor Technologies and Applications in Agriculture and Food Industry: State of the Art and Current Trends. *Sensors*, 2009.

[46] A. R. Schepers, J. F. Shanahan, M. A. Liebig, J. S. Schepers, S. H. Johnson, and A. Luchiari. Appropriateness of Management Zones for Characterizing Spatial Variability of Soil Properties and Irrigated Corn Yields across Years. *Agronomy Journal*, 2004.

[47] D. Schimmelpfennig. Cost Savings From Precision Agriculture Technologies on U.S. Corn Farms. *USDA Amber Waves*, 2016.

[48] SenseFly. Sensefly. sensefly.com.

[49] Sentera. https://sentera.com/.

[50] M. Shamos. *Computational Geometry*. Yale University, 1978.

[51] N. Sharma, J. Gummeson, D. E. Irwin, and P. J. Shenoy. Cloudy Computing: Leveraging Weather Forecasts in Energy Harvesting Sensor Systems. In *IEEE SECON*, 2010.

[52] SIGFOX. http://www.sigfox.com/.

[53] S. N. Sinha and M. Pollefeys. Pan–tilt–zoom Camera Calibration and High-resolution Mosaic Generation. *Computer Vision and Image Understanding*, 2006.

[54] M. Soderstrom, G. Sohlenius, L. Rodhe, and K. Piikki. Adaptation of Regional digital soil mapping for precision agriculture. *Precision Agriculture*, 2016.

[55] J. V. Stafford. *Precision Agriculture Book*. 2005.

[56] R. Szeliski. Image Alignment and Stitching: A Tutorial. Technical report, Microsoft Research, 2004.

[57] W. T., P. Csiro, P. Corke, Sikka, L. Klingbeil, Y. Guo, C. Crossman, P. Valencia, D. Swain, and G. Bishop-Hurley. Transforming Agriculture through Pervasive Wireless Sensor Networks. *IEEE Pervasive Computing*, 2007.

[58] United Nations General Assembly. Food Production Must Double by 2050 to Meet Demand from Worlds Growing Population, Innovative Strategies Needed to Combat Hunger, Experts Tell Second Committee, 2009. http://www.un.org/press/en/2009/gaef3242.doc.htm.

[59] C. M. Vigorito, D. Ganesan, and A. G. Barto. Adaptive Control of Duty Cycling in Energy-harvesting Wireless Sensor Networks. In *IEEE SECON*, 2007.

[60] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and Yield in a Volcano Monitoring Sensor Network. In *Operating Systems Design and Implementation*, 2006.

10 Supplementary Material

10.1 FarmBeats Applications and Usage

The goal of FarmBeats is to serve as a substrate for multiple sensing modalities on the farm. Sensors with varying data requirements can plug-in to FarmBeats and operate seamlessly. This allows farmers to use FarmBeats for various applications. Our primary target has been to deliver a class of applications that fall under the category of precision agriculture. Precision agriculture is a technique to improve yield by treating the farm as heterogeneous land, and uses variable treatment throughout the farm, such as variable seeding, fertilizer application, lime application, irrigation, and many other agricultural services. In principle, precision agriculture is good for the overall farming ecosystem [37]. It improves yield, reduces the operating expenses for the farmer [47], and is also good for the environment.

In contrast to existing systems which divide the farm into large static management zones [14, 41] and fail to capture temporal and climatic variations [32, 46], FarmBeats delivers near real-time and fine-grained precision maps to farmers for soil characteristics like pH, moisture, etc. As described in section 5.3, FarmBeats uses a combination of aerial imagery and sparse sensor deployment to deliver these maps. They are currently being used by farmers to monitor and amend irrigation, and lime application practices in their farms.

In addition to precision agriculture applications, farmers have used FarmBeats for monitoring cattle using cameras in barns and for using sensors to monitor temperature in storage units. This functionality was suggested by farmers and later added to the system. Future applications suggested by farmers include monitoring the net carbon footprint of the agricultural production cycle, net nutrient usage for each crop cycle, crop suggestions using long-term data and flood monitoring.

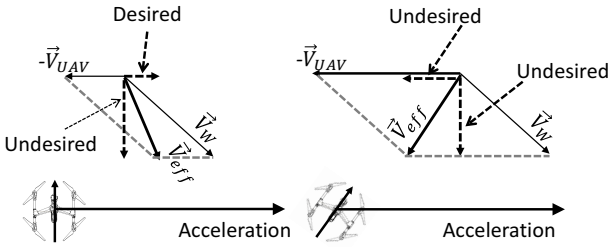


Figure 10: **Leveraging Wind Assistance:** When the effective velocity, $\vec{v}_{eff}$, aids the UAV motion, then the yaw is set to be perpendicular to the UAV motion (left), otherwise the yaw is aligned with the effective wind velocity to minimize air drag (right).

10.2 Leveraging Wind to Assist UAV Path Planning

Algorithm 1 Pseudo-code for determining UAV yaw based on wind speed

▷ Input: Wind velocity ($\vec{v}_W$), UAV velocity ($\vec{v}_{UAV}$), intended acceleration ($\vec{a}_{UAV}$)
▷ where all inputs are measured with respect to the earth reference frame
▷ Output: UAV yaw (y)
Compute wind velocity w.r.t the UAV: $\vec{v}_{eff} = \vec{v}_W - \vec{v}_{UAV}$
if $\vec{v}_{eff}.\vec{a}_{UAV} > 0$ **then**
    Set the yaw perpendicular to $\vec{a}_{UAV}$
    $y = \angle\vec{a}_{UAV} + \frac{\pi}{2}$
else
    Align the yaw with $\vec{v}_{eff}$ to minimize drag
    $y = \angle\vec{v}_{eff}$
end if

Here, we describe the algorithm to leverage wind to as-

sist in UAV path planning. On a high level, the algorithm works in two steps:

- Compute the effective wind velocity, $\vec{v}_{eff}$. The effective wind velocity is the wind velocity in the reference frame of the UAV. Specifically, if the velocity of the UAV with respect to the ground is $\vec{v}_{UAV}$ and the wind velocity with respect to the ground is $\vec{v}_W$, then $\vec{v}_{eff} = \vec{v}_W - \vec{v}_{UAV}$. An example of this vector computation is shown in figure 10.
- If $\vec{v}_{eff}$ has a component that can aid the UAV motion, then we make the yaw perpendicular to the direction of intended acceleration, otherwise, we align it with the direction of $\vec{v}_{eff}$ to minimize air drag. For instance, as shown in figure 10, if the UAV wants to accelerate and the UAV velocity is large such that the effective wind velocity has no component that aids acceleration, then the algorithm aligns the UAV yaw with the effective wind velocity, minimizing the air drag.

Algorithm 1 describes the pseudo-code for FarmBeats's approach.

Enabling High-Quality Untethered Virtual Reality

Omid Abari Dinesh Bharadia Austin Duffield Dina Katabi
Massachusetts Institute of Technology

Abstract

Today's virtual reality (VR) headsets require a cable connection to a PC or game console. This cable significantly limits the player's mobility and, hence, her VR experience. The high data rate requirement of this link (multiple Gbps) precludes its replacement by WiFi. Thus, in this paper, we focus on using mmWave technology to deliver multi-Gbps wireless communication between VR headsets and their game consoles. We address the two key problems that prevent existing mmWave links from being used in VR systems. First, mmWave signals suffer from a blockage problem, i.e., they operate mainly in line-of-sight and can be blocked by simple obstacles such as the player lifting her hand in front of the headset. Second, mmWave radios use highly directional antennas with very narrow beams; they work only when the transmitter's beam is aligned with the receiver's beam. Any small movement of the headset can break the alignment and stall the data stream. We present MoVR, a novel system that allows mmWave links to sustain high data rates even in the presence of a blockage and mobility. MoVR does this by introducing a smart mmWave mirror and leveraging VR headset tracking information. We implement MoVR and empirically demonstrate its performance using an HTC VR headset.

1 Introduction

The past few years have witnessed major advances in augmented reality and virtual reality (VR) systems, which have led to accelerated market growth. Facebook has recently started shipping their VR headset (Oculus Rift) and expects to ship more than 2 million headsets by 2017 [6]. HTC sold more than 15,000 VR headsets in the first 10 minutes following their release [8]. These devices are expected to soon dominate the gaming and entertainment industry, and they have found applications in manufacturing and healthcare [20, 21]. However, a key challenge prevents this technology from achieving its full potential. High-quality VR systems need to stream multiple Gbps of data from their data source (PC or game console) to the headset. As a result, these headsets have an HDMI cable snaking down the player's neck and hardwiring her to the PC, as shown in Fig 1. The cable not only limits the player's mobility and interferes with the VR experience, but also creates a tripping hazard since the headset covers the player's eyes. This has left the industry searching for untethered solutions that can deliver a high-quality VR experience without these limitations. Unfortunately, typical wireless systems, such as WiFi, cannot support the required data rates. This challenge has led to awkward products: Zotac has gone as far as stuffing a full PC in the player's backpack in the hope of delivering an untethered VR.

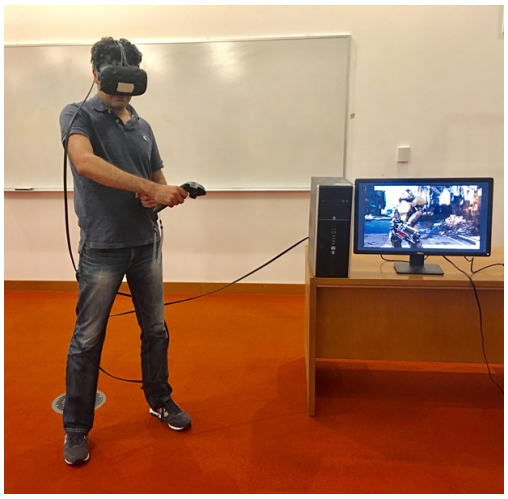


Figure 1—Virtual Reality experience: The figure shows the headset's cable not only limits the player's mobility but also creates a tripping hazard.

Ideally, one would like to replace the HDMI cable with a wireless link. Thus, multiple companies have advocated the use of mmWave for VR since mmWave radios have been specifically designed to deliver multi-Gbps data rates [16, 1]. The term mmWave refers to high frequency RF signals in the range of 24 GHz and higher [15, 7]. The 802.11ad standard operates in mmWave and can transmit over 2 GHz of bandwidth and deliver up to 6.8 Gbps. No other consumer RF technology can deliver such data rates. However, mmWave links bring up new challenges that must be addressed before this technology can be used for VR applications:

- **Dealing with blockage:** mmWave links require a line-of-sight between transmitter and receiver, and they do not work well through obstacles or reflections. This problem is due to the fact that mmWave antennas are highly directional and typically generate narrow beams. Hence, even a small obstacle like the player's

hand can block the signal. Said differently, these links work well when the receiver on the headset has a clear line-of-sight to the transmitter connected to the PC, but if the player moves her hand in front of the headset (see Fig. 3), or other people in the environment obstruct the receiver's view to the transmitter, the signal will be temporarily lost, causing a glitch in the data stream (shown in our empirical results in §4). While temporary outages are common in wireless communication, VR data is non-elastic, and cannot tolerate any degradation in SNR and data rate.

- **Dealing with mobility:** Since mmWave radios use highly directional antennas, they work only when the transmitter's beam is aligned with the receiver's beam. Further, since the wavelength is very small, even a small movement of the headset can hamper the alignment and break the link. Past work on mmWave typically assumes static links and, hence, fixed alignment [39, 48, 32]. Identifying the correct alignment for the antennas can take up to multiple seconds [49, 44]. Such delay is unacceptable for VR systems, which need to play a new frame every 10 milliseconds, even when the headset moves [10].

This paper introduces MoVR, a wireless system that enables a reliable and high-quality untethered VR experience via mmWave links. MoVR addresses the main challenges facing existing mmWave links. In particular, MoVR overcomes the blockage problem by introducing a self-configurable mmWave mirror that detects the incoming signal and reconfigures itself to reflect it toward the receiver on the headset. In contrast to a traditional mirror, a MoVR mirror does not require the angle of reflection to be equal to the angle of incidence. Both angles can be programmed so that our mirror can receive the signal from the mmWave transmitter attached to the data source and reflect it towards the player's headset, regardless of its direction. In §4.2, we explain the design of such mmWave mirrors and how they can be implemented simply by deflecting the analog signal without any decoding.

Next, MoVR ensures that the VR system sustains high data rates to the headset in the presence of mobility. In contrast to past work on mmWave [1, 26, 49], MoVR does not scan the space to find the best way to align the mmWave directional antennas, a process known to incur significant delay [49, 44]. Specifically, MoVR finds the best beam alignment by relying on existing tracking functions available in VR systems. In designing MoVR, we observe that VR systems already track the location of the headset to update the 3D view of the player. Thus, MoVR leverages this information to quickly localize the headset and move the transmitter antenna's beam with it. However, while the VR application tracks the movements of the headset, it does not know the location of the headset with respect to the mmWave transmitter and the MoVR mirror. Thus, we design a novel algorithm that combines the output of VR headset tracking with RF-based localization to quickly steer the mmWave antennas and keep their beams aligned as the player moves around.

We have built a prototype of MoVR and evaluated its performance empirically using an HTC VR system. Our results can be summarized as follows:

- In the absence of MoVR's mirror, even a small obstacle like the player's hand can block the mmWave signal and result in a drop in SNR of 20dB, leaving the VR headset with no connectivity. The addition of MoVR's mirror prevents the loss of SNR in the presence of blockage, sustaining high data rates.

- Given the VR headset information, MoVR aligns the antenna beams in under a few micro seconds, which is negligible compared to the user's movement. Further, the resulting alignment sustains the required high SNR and VR data rates.

- Finally, in a representative VR gaming setup, MoVR provides an SNR of 24dB or more for all locations in the room and all orientations of the headset, even in the presence of blockage and player mobility. This SNR is much higher than the 20dB needed for the VR application.

2 Related Work

Related work can be classified into three areas.

(a) Virtual Reality: Existing VR systems can be divided into PC-based VR, like Occulus Rift and HTC Vive, and Gear VR, like systems by Samsung and Visus [17, 22]. PC-based VR systems leverage their computational horsepower to generate rich graphics that look realistic and support fast head motion. They require, however, an HDMI cable to connect the PC to the headset. Gear VR slides a powerful smart phone into the headset. Thus, they do not need an external cable. Their mobility, however, is limited by the inability to support rich graphics that react to motion; their imagery tends to blur with motion [3]. There is a huge interest in untethered PC-based VR systems. Optoma and SiBeam have proposed using mmWave radios to connect the headset to the PC, but they have not provided any details about their proposal [16, 1]. Sulon proposed to equip the headset with an integrated computer [18]. Unfortunately, this would make the headset much larger and heavier, thus interfering with the user experience. WorldViz advertises a wireless wide-area tracking system. However, they still require the user to have a cable for the display or carry

a limited data source and a processor unit [25]. Zotac advertises a mobile VR headsets where the user carries the PC in a backpack. Finally, Google has recently announced that their next VR headset will be wireless, but has not provided any details of the design or the release date [23].

(b) mmWave Communications: Much past work on mmWave communication addresses *static links*, such as those inside a data center [39, 48, 32], where there is a line-of-sight path between the transmitter and receiver. Some past work looks at mobile links for cellular networks or wireless LANs [46, 43, 40]. These systems typically scan the space to align the antennas, a process that takes up to several seconds [49, 44]. In contrast, by leveraging the fact that VR systems already track the headset, MoVR is able to speed up antenna steering enough that it can be done faster than the VR frame rate, as we show in §7. Also, most past work on mmWave links assumes line-of-sight connectivity. Some papers do consider scenarios in which the line-of-sight between transmitter and receiver is blocked [46, 45, 38]. However, since they target elastic applications, their solution switches the directional antenna to the best reflected path, which typically has a much lower SNR (see Fig. 4). In contrast, the VR application is non-elastic and cannot tolerate reduction in its SNR and data rate. Also, there are wireless HDMI (WHDI) products from LG and Samsung which operate at mmWave frequencies, but these products assume static links and require line-of-sight between the receiver and transmitter [24]. Thus, they cannot adapt their direction and will be disconnected if the player moves. Finally, the work in [48] has proposed a form of mmWave mirror to reflect an RF signal off the ceiling of a data center. Their approach, however, covers the ceiling with metal. Such a design is unsuitable for home applications and cannot deal with player mobility.

(c) Relay and Full-Duplex: The design of MoVR mirrors is related to that of wireless relays at lower frequencies (e.g., Wi-Fi and LTE [19]). Similarly to a MoVR mirror, these relays amplify and forward the signal of interest. However, they do not deal with the issue of directionality. In contrast, our MoVR mirror needs to capture the mmWave signal along a particular direction and reflect it in the direction of the headset. Finally, MoVR mirrors are related to previous work on full-duplex relays since they receive a signal and transmit it at the same time. However, full-duplex radios require complex analog and digital hardware with full transmit and receive chains [29]. In contrast, MoVR mirrors have only an analog front-end (i.e., antennas and an amplifier) and do not need digital transmit or receive chains.

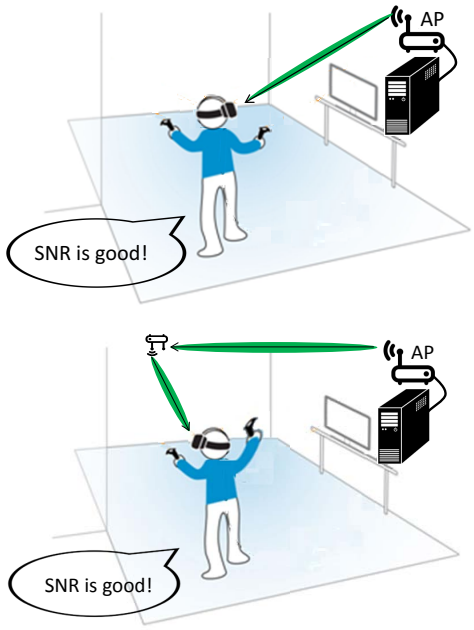


Figure 2—MoVR's setup: The figure shows MoVR's setup. The PC is connected to a mmWave AP and the headset is equipped with a mmWave receiver. In the case of a blockage (e.g., the user raises her hand or turns her head), the AP delivers its signal by reflecting it off a MoVR mirror.

3 MoVR Overview

MoVR is a wireless communication system for virtual reality applications. It enables a sustainable, high-data-rate wireless link even in the presence of blockage and headset mobility. High-quality VR systems stream multiple Gbps from a high-power PC to the headset. MoVR delivers this data over a mmWave wireless link. Fig. 2 shows MoVR's setup. The PC is connected to a mmWave transmitter, which we refer to as the AP, and the headset is equipped with a mmWave receiver. As the figure shows, MoVR operates in two modes, depending on the real-time scenario: when the direct path from the AP to the headset is clear, the AP beams its signal to the headset. However, if the direct path is blocked, the AP detects the blocking and reflects its signal to the headset via a MoVR mirror. The environment may have one or more MoVR mirrors; the AP picks the best one depending on the headset location.

The next few sections present the components that contribute to the design of MoVR. We start by explaining the two key challenges in using mmWave links in VR systems, and how we overcome them. We then explain how the various components work together to satisfy the application.

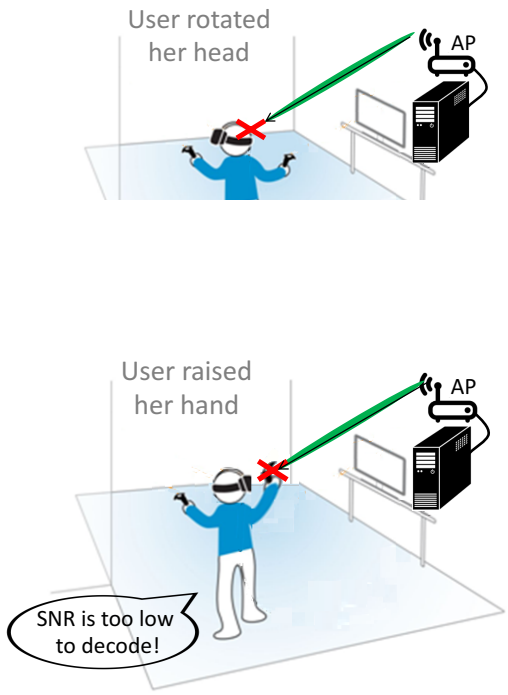


Figure 3—Blockage Scenarios: As the user moves his head or hand, the line-of-sight path between the AP and the headset's receiver can be easily blocked. This results in a significant drop in SNR and data rate

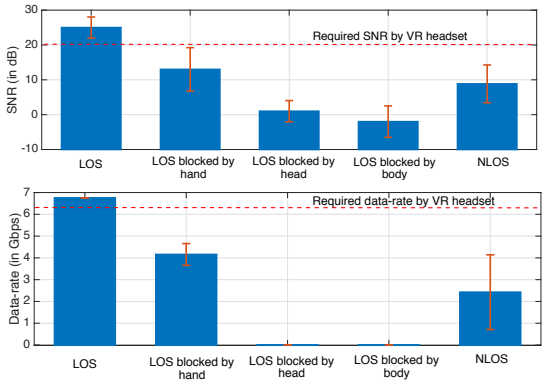


Figure 4—Blockage impact on data rate. The Figure shows SNR and data rate for different scenarios: line-of-sight (LOS) without any blockage, LOS with different blockages and non-line-of-sight (NLOS). The figure shows that blocking the signal with one's hand, head, or body results in a significant drop in SNR and causes the system to fail to support the required VR data rate. The figure also shows that simply relying on NLOS reflections in the environment does not deliver good SNR and would fail to support the required data rate.

4 Blockage Problem

A key challenge in using mmWave links for VR applications is that they may be easily blocked by a small obstacle, such as the player's hand. This is a side effect of highly directional antennas, which mmWave radios must use to focus their power and compensate for path loss. Below, we investigate the impact of blockage in more detail, then explain our solution to overcome this problem.

4.1 Impact of Blockage

We first investigate the impact of blocking the direct line-of-sight on the signal's SNR and the link's data rate. To do so, we attach a mmWave radio to an HTC PC-based VR system and another one to the headset (see §7 for hardware details). We conduct experiments in a $5m \times 5m$ room. We place the headset in a random location that has a line-of-sight to the transmitter, and measure the SNR at the headset receiver. We then block the line-of-sight and measure the SNR again. We consider different blocking scenarios: blocking with the player's hand, blocking with the player's head, and blocking by having another person walk between headset and the transmitter. We repeat these measurements for multiple different locations. Fig. 4 shows the results of this experiment: the top graph shows the SNR and the bottom graph shows the data rate. The SNRs are measured empirically and the corresponding data rates are computed by substituting the SNR measurements into standard rate tables based on the 802.11ad modulation and code rates [12, 13, 9]. The first bar in Fig. 4 shows that, in the absence of blocking, the mean SNR is 25dB and the resulting data rate is almost 7 Gbps, which exceeds the needs of the VR application. Bars 2, 3, and 4 in the figure correspond to different blocking scenarios. They show that even blocking the signal with one's hand degrades the SNR by more than 14 dB and causes the data rate to fail to support the VR application.

One solution to overcome this challenge is to rely on non-line-of-sight paths –i.e., the signal reflections from walls or other objects in the environment. For example, both the transmitter and the headset receiver can direct their signal beams toward a wall and rely on the natural reflection from the wall. In fact, this is how current mmWave systems work. Unfortunately, non-line-of-sight paths typically have much higher attenuation than the line-of-sight path due to the fact that walls are not perfect reflectors and therefore scatter and attenuate the signal significantly. Moreover, signals travel a longer distance in non-line-of sight scenarios than in line-of sight scenarios, which results in higher attenuation.

To confirm, we repeat the measurements for all blocking scenarios, but instead of trying to receive the signal along the blocked direct path, we sweep the mmWave beam on the transmitter and receiver in all directions. We try every combination of beam angle for both transmitter and receiver antennas, with 1 degree increments. We ignore the direction of the line-of-sight and note the maximum SNR across all non-line-of-sight paths. The

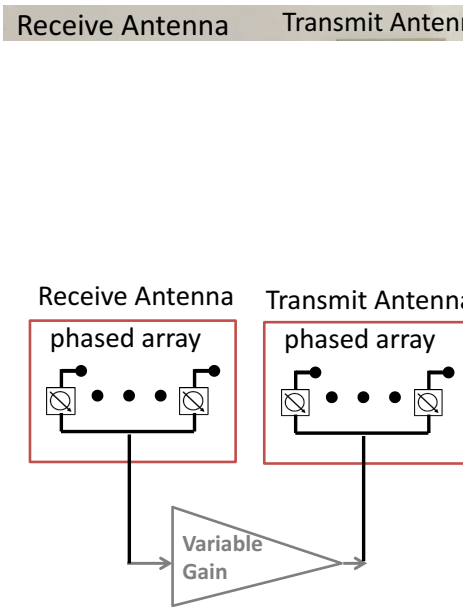


Figure 5—MoVR programmable mmWave mirror: The figure shows (a) the implementation and (b) the block diagram of the mirror. The design of the mirror is small and simple. It consists of two directional phased-array antennas connected via a variable-gain amplifier.

last bar in Fig. 4 shows the results of this experiment. It shows that when the transmitter and receiver have to use a non-line-of-sight path, the SNR drops by 16dB on average. The figure also shows that this reduction in SNR causes the data rate to fail to support the VR application.

Note that one cannot solve the blockage problem by putting more antennas on the back or side of the headset, since the line-of-sight from the AP to the headset may get completely blocked by the player's hands or body (as shown in Fig. 3), or by the furniture and other people in the environment. One naïve solution to overcome this challenge is to deploy multiple mmWave APs in the room to guarantee that there is always a line of sight between the transmitter and the headset receiver. Such a solution requires extending many HDMI cables in the environment to connect each AP to the PC. However, this defeats the purpose of a wireless design because it requires enormous cabling complexity. Further, requiring multiple full-fledged mmWave transceivers will significantly increase the cost of VR systems and limit their adoption in the market.

In the next section, we describe how to overcome the blockage problem using mmWave mirrors. Our mirrors do not need to connect to the PC, and have a simple cheap design with no digital transmit or receive components.

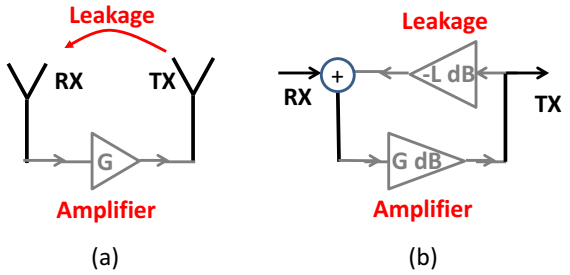


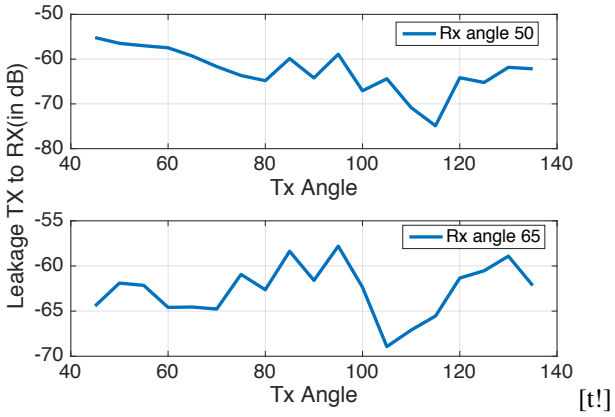
Figure 6—MoVR's mirror block diagram: The figure shows (a) the block diagram and (b) equivalent signal-flowgraph of MoVR's mirror. The figure shows that the input signal is first amplified by G_{dB}, then attenuated by L_{dB} and fed back to the input as a leakage.

4.2 Programmable mmWave Mirrors

To overcome the blockage problem, we designed a programmable mmWave mirror that can control both the angles of incidence and reflection [27]. Fig. 5 shows a basic diagram of the circuit and a picture of our prototype. Each MoVR mirror consists of a transmit and receive antenna connected via a variable-gain amplifier. As is common in mmWave radios, the antennas are implemented using phased arrays in order to create highly-directional beams, which can be steered electronically in a few microseconds. Note that the design is quite simple. Specifically, it neither decodes the signal nor includes any transmit or receive digital components (DAC, ADC, mixer, etc.). This allows us to avoid complex and expensive components that would have to operate at multiple Gbps.

An important challenge in designing such a mmWave mirror stems from the leakage between the transmit and receive antennas. At a high level, a MoVR mirror works by capturing the RF signal on its receive antenna, amplifying it, and reflecting it using a transmit antenna. However, some of the signal reflected by the mirror is also received by its own receive antenna. This means that the output of the amplifier is fed back to the input of the amplifier. This creates a feedback loop that can cause the amplifier to saturate, thereby generating garbage signals. Thus, a key question in designing MoVR mirrors is: how do we set the optimal amplifier gain so that we avoid saturation, but also maximize the SNR delivered to the headset?

In order to ensure that the leaked signal is damped while the signal of interest (i.e., the received signal from the AP) is amplified, we need to ensure that the amplifier gain is less than the leakage. To see why this is the case, consider the signal-flow-graph of the mirror, shown in Fig. 6(b). The input signal is first amplified by G_{dB}, then attenuated by L_{dB} and fed back to the input. From Control Theory [33], we know that for this system to stay stable we need to ensure that $G_{dB} - L_{dB} < 0$ [33, 30]. This implies that the amplifier gain (G_{dB}) must be lower



[t!]

Figure 7—Leakage between mirror's transmit and receive antennas: The figure shows the leakage across different transmit beam directions for two different receive beam directions. The figure shows that the leakage variation can be as high as 20dB. This result confirms a need for an adaptive algorithm that reacts to the leakage in real time and adjusts the amplifier gain accordingly.

than the absolute value of the leakage (L_{dB}); otherwise the system becomes unstable, leading to saturation of the amplifier.

To avoid this saturation, the mirror needs to measure the leakage and then set the amplification gain lower than the leakage. The leakage, however, varies when the direction of the antenna beam changes to track the headset. Fig. 7 shows the leakage across different transmit beam directions for two different receive beam directions. As we can see, the leakage variation can be as high as 20dB. The variation of the leakage and the fact that the amplifier gain must always be set lower than the leakage create a need for an adaptive algorithm that reacts to the leakage in real time and adjusts the amplifier gain accordingly.

One naïve algorithm is to send a signal from the mirror's transmit antenna and measure the received power at its receive antenna in order to estimate the amount of leakage, then to use this information to set the amplifier gain accordingly. However, we cannot do this since a MoVR mirror does not have digital transmit and receive chains.

Our solution exploits a key characteristic of amplifiers: an amplifier draws significantly higher current (from a DC power supply) as it gets close to saturation mode, compared to during normal operation [37, 31].[1] We can therefore detect if the amplifier is getting close to its saturation mode by monitoring the current consumption from the power supply. Thus, our gain control algorithm works as follows: it sets the amplifier gain to the minimum. It increases the amplifier gain step by step while monitoring the amplifier's current consumption. The algorithm continues increasing the gain until the current consumption suddenly goes high. This indicates that the amplifier is entering its saturation mode. The algorithm then backs off, keeping the amplification gain just below this point.

[1]The exact quantity of the amplifier's current consumption for its different operating modes are specified in its datasheet. We use a simple IC which measures the current consumption of the amplifier to detect its operating mode.

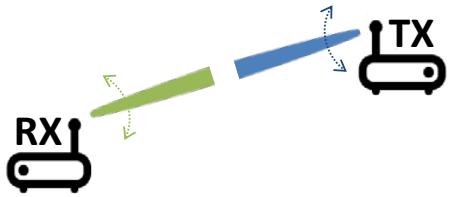


Figure 8—Beam alignment in mmWave radios: mmWave radios need to find the best alignment between the transmitter's and receiver's beams to establish a communication link.

5 Dealing with Mobility

Movement of the VR headset creates a critical challenge for mmWave links. Specifically, mmWave frequencies suffer from a large path loss. To compensate for this loss, mmWave radios use highly directional antennas to focus the signal power in a narrow beam. Such directional antennas can be implemented using phased arrays. In fact, since the wavelength is very small (on the order of a millimeter), tens or hundreds of such antennas can be packed into a small space, creating a pencil-beam antenna. The beam can be steered electronically in a few microseconds. However, the real challenge is to identify the correct spatial direction that aligns the transmitter's beam with the receiver's beam (as shown in Fig. 8). This is particularly difficult in VR applications since the headset is naturally in a mobile state.

Below, we investigate the impact of beam misalignment on the signal and explain our solution to overcome this problem.

5.1 Impact of Beam Misalignment

We first investigate the impact of beam misalignment on the SNR of the signal delivered to the headset. To do so, we attach a mmWave transmitter to the VR PC (which we call the AP), and a mmWave receiver to the headset. We position the headset's receiver such that it has a line-of-sight path to the AP's transmitter. We ensure that the transmitter's and receiver's beams are perfectly aligned by scanning for all possible alignments and picking the one that maximizes the SNR. We use this setup as the initial position of the headset in our experiment –i.e., we start the experiment with a perfect beam alignment. We

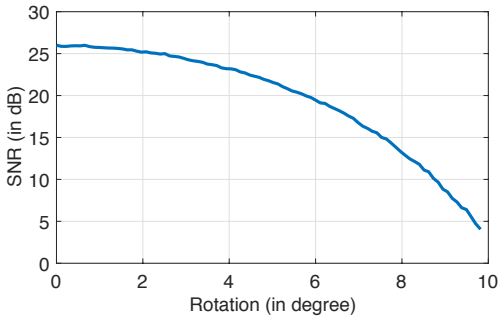


Figure 9—SNR versus amount of headset rotation: Even a minor head rotation of a few degrees can cause a major degradation in the received SNR. This result confirms the need for real-time beam tracking to realign the transmitter's and the receiver's beams as the player moves her head.

then rotate the headset and measure the SNR as a function of the angluar deviation from the perfect orientation. Note that the headset rotation causes misalignment between the transmitter's and receiver's beams. Fig. 9 shows the SNR of the received signal versus the amount of headset rotation. The figure shows that even a minor head rotation of a few degrees can cause a major degradation in the SNR. As was shown in §4, such reduction in SNR creates outages for the VR application. This experiment confirms the need for real-time beam tracking to realign the transmitter's and the receiver's beams as the player moves her head.

5.2 Beam Alignment and Tracking

In this section, we explain how MoVR aligns the transmitter's and receiver's beams, and adapts the alignment as the headset moves. Recall that MoVR operates in two different modes (as shown in Fig. 2). In the first mode, the AP communicates to the headset directly. This requires a beam alignment between the AP and the headset. In the second mode, the AP communicates to the headset through the mirror. This requires beam alignment between the AP and the mirror, and beam alignment between the mirror and the headset. Therefore, there are three types of beam alignment which need to be addressed in MoVR. Below, we explain each of them in more detail.

(a) Beam alignment between the AP and the mirror: To deliver the signal from the AP to the mirror, the AP needs to align its transmit beam toward the mirror and the mirror needs to align its receive beam toward the AP. Since both the AP and the mirror are static, this alignment is only done once when the mirror is installed. Though this alignment has no real-time constraints, it cannot employ past work on beam alignment [47, 41, 34, 42, 36] because all of these schemes require both nodes to transmit and/or receive signals. A MoVR mirror, however, can neither transmit nor receive; it can only reflect signals.

Thus, MoVR delegates to the AP the task of measuring the best beam angle, which the AP can then communicate to the mirror using a low-bit-rate radio, such as bluetooth. During this estimation process, the AP transmits a signal and the mirror tries to reflect this signal back to the AP itself (instead of reflecting it to the headset) allowing the AP to measure the best angle. The mirror, however, does not yet know the direction of the AP, so it has to try various directions and let the AP figure out the direction that maximizes the SNR.

Thus, our algorithm works as follows. It first sets the mirror's receive and transmit beams to the same direction, say θ_1, and sets the AP's receive and transmit beams to the same direction, say θ_2. Then it tries every possible combination of θ_1 and θ_2 while the AP is transmitting a signal and measuring the power of the reflection (from the mirror). The θ_1 and θ_2 combination that gives the highest reflected power corresponds to the angles for the best alignment of the mirror's receive beam and the AP's transmit beam.

One problem remains. As described above, the AP needs to measure the power of the signal reflected by the mirror, while transmitting its own signal. Performing this measurement is not easy. This is due to the fact that the AP is trying to transmit and receive at the same time. As a result, the transmitted signal leaks from the AP's transmit antenna to its receive antenna. So to measure the reflected signal power, the AP first needs to separate it from the strong leakage signal it receives.

To overcome this problem, we use the fact that, if the mirror modulates the signal before it reflects it, the AP can separate the reflected signal from the leakage signal as the two signals become different. For example, if the AP transmits a sinewave at a frequency f_1, and the mirror modulates this signal by turning its amplifier on and off at a frequency f_2, then the center frequency of the reflected signal will be $f_1 + f_2$ while the leakage signal remains at f_1. Hence, the AP can simply use a filter to separate the reflected signal from the leakage signal.

(b) Beam alignment and tracking between the AP and the headset: To deliver the signal directly from the AP to the headset, the AP needs to align its transmit beam toward the headset and the headset needs to align its receive beam towards the AP. Here, we will explain how MoVR estimates the angles for best alignment of the AP's transmit beam and headset's receive beam.

We observe that VR systems have to track the location and orientation of the headset in order to update the 3D view of the player. Specifically, the HTC VR system does this using laser trackers and an IMU on the headset. Using this infrastructure, the VR system is able to calculate the headset's exact position relative to each

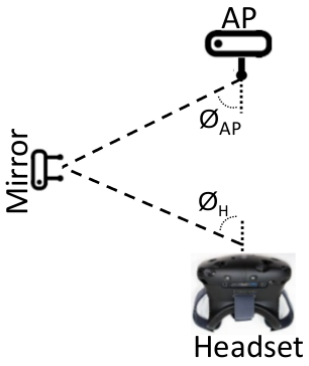


Figure 10—Localizing the mirror: MoVR finds the location of the mirror in the VR setup by intersecting the line-of-sight angle from the AP to mirror (ϕ_{AP}) and the line-of-sight angle from the headset to the mirror (ϕ_H).

laser tracker. By co-locating MoVR's AP with one of the VR's laser trackers, we can exploit the VR tracking system to find the exact location and orientation of the headset relative to the AP.[2] MoVR leverages this information to calculate the best alignment to the headset and track the alignment in real time.

(c) Beam alignment and tracking between the mirror and the headset: In order to align the mirror's transmit beam with the headset's receive beam, we need to know the location and orientation of the headset with respect to the mirror. Unfortunately, the VR tracking system only has this information with respect to the AP. In order to switch the reference point, we need to get the location and orientation of the mirror with respect to the AP. Getting the orientation was explained in §5.2(a), but we still need to get the distance between the AP and the mirror.

Because the location of the mirror with respect to the AP is fixed during use of the VR system, one naïve solution is to ask the user to measure it during installation. However, this requires an accurate measurement, since even a small measurement error creates a significant inaccuracy in beam alignment. To avoid this, MoVR uses an automated calibration mechanism which calculates the location of the mirror with respect to the AP without any help from the user.

This calibration mechanism works by intersecting the line-of-sight angle from the AP to the mirror (ϕ_{AP}) and the line-of-sight angle from the headset to the mirror (ϕ_H), as shown in Fig. 10. In §5.2(a), we explained how MoVR estimates ϕ_{AP}. To estimate ϕ_H, MoVR first configures the AP to transmit to the mirror. Then it tries every combination of mirror transmit beam angle and headset receive beam angle. The receive beam angle which gives the highest SNR at the headset corresponds to ϕ_H. Finally, by intersecting the two spatial directions ϕ_H and ϕ_{AP}, we can determine the location of the mirror. Because the mirror location is fixed, this process only needs to happen once, during installation. Subsequently, MoVR can calculate the beam alignment between the mirror and the headset from VR tracking information and the mirror's known location.

[2] In practice, location of the AP may be a few cm different from the location of the laser tracker. Since this is a fixed deviation, it can be calibrated by the manufacturer. Also note that even if the AP is blocked from the headset, the VR system has enough redundancy to localize the headset.

6 System Details

The last sections presented the solutions to the two challenges in using mmWave in VR systems –i.e. blockage and mobility. However, a number of system details must be addressed in order to put these solutions into practice. In particular: How do we provide connectivity at all locations within the VR space? And how do we choose between sending data via the direct link or sending by way of the mirror? This section will iron out these details and provide guidance on the system design trade-offs to maximize MoVR's coverage and performance.

How do we provide connectivity at all locations within the VR space? The system's ability to provide wireless coverage throughout the VR space is limited by the fact that the line-of-sight to the headset can be blocked by the environment and/or by the user's limbs. We addressed this problem by designing the MoVR mirror described in §4. However, it is still possible that the headset experiences blocking along the path to both the AP and to the mirror. To address this issue, MoVR supports multiple mirrors. Each mirror adds another path to the headset, and reduces the probability of the headset being blocked exponentially. Additionally, we recommend placing multiple antennas on the headset to reduce the probability of the user's head blocking a line-of-sight path. However, any body parts that come between the headset and the AP can block all headset antennas, necessitating the use of a mirror.

How do we choose between the direct link and a link via a mirror? The AP, the mirrors, and the headset are equipped with a cheap, low-bitrate radio, e.g. Bluetooth, to exchange control information. In MoVR, the mmWave receiver on the headset continuously monitors the SNR of its received signal and, whenever it drops below a certain threshold, the headset reports it back to the AP over Bluetooth. The AP then switches to a different link. The AP picks the mirror closest to the current location of the headset. If the SNR does not go above the desired threshold, the AP switches to the next closest mirror.

Figure 11—MoVR mirror's controller board: The figures shows our custom-designed controller board for configuring the beam alignment and the amplifier gain in real time.

Because any small period of outage impacts the quality of the data rate, the headset should act preemptively by looking at the time series of SNR and ordering a link change if there is a downward trend that is likely to result in outage. As demonstrated in §7, the switching latency cost is sufficiently small that it does not impact the user experience, even if the AP tries more than one mirror.

7 Evaluation

We have built a prototype of MoVR using off-the-shelf components. MoVR's mirror hardware consists of two phased array antennas (one for receive and one for transmit), connected to each other through a variable gain amplifier as shown in Fig. 5. The phased arrays consist of patch antenna elements, designed and fabricated on PCB. The outputs of the patch antennas are connected to Hittite HMC-933 analog phase shifters, which allows us to steer the antennas' beams. To create a variable gain amplifier, we use a Hittite HMC-C020 PA, a Quinstar QLW-2440 LNA and a Hittite HMC712LP3C voltage-variable attenuator. The mirror's current consumption is mainly dominated by it's PA, which consumes 250mA during normal operation. For controlling MoVR's mirror and measuring its amplifier's current consumption, we built a controller board using an Arduino Due micro-controller, Analog Devices DACs, and a Texas Instruments INA169 DC current sensor, as shown in Fig. 11.

In our experiments, we use the HTC VIVE virtual reality system. However, our design is also compatible with other high-quality VR platforms, such as Oculus Rift. [3] We equip the VIVE headset with a mmWave receiver and the VR PC with a mmWave AP working at the 24GHz ISM band [28]. The PC has an Intel i7 processor, 16GB RAM and a GeForce GTX 970 graphics card, which is required for the HTC VIVE VR setup. The transmission power is in accordance with FCC rules [35].

We evaluate MoVR in a $5m \times 5m$ office room with standard furniture, such as desks, chairs, computers and closets.[4] We perform experiments in both line-of-sight and non-line-of sight scenarios.

7.1 Blockage During an Actual VR Game

In §4, we demonstrated the impact of signal blockage on the SNR and data rate of mmWave communications in a VR setup. As was shown, even blocking the signal with one's hand significantly degrades the data rate, which is problematic. In this experiment, we investigate how often the AP's line-of-sight to the headset is blocked in a realistic gaming scenario.

To do so, we ask a user to play a VR game (The Lab Solar System) while we extract the location information of the headset, access point, and two game controllers (held by the player's hands) from the VR tracking system. Using this information, we find the equation of the line between the headset and the base station. Then, as the user plays the game, we check if the locations of the user's hands (i.e. controllers' locations) lie on that line. If either hand lies on the line, we conclude that the line-of-sight path between the headset and the base station is blocked. Our results show that the line-of-sight was blocked 20 times during a 5 minute game. Fig. 12 plots the CDF of the durations of these 20 cases. The figure shows that the median blockage duration is 245ms. Note that the VR frame rate is only 10ms. Hence, 245ms of blockage is highly detrimental to the user's VR experience. Our results confirm that line-of-sight blockages happen very often during VR games and persist long enough to degrade the VR experience.

7.2 MoVR's Mirror Performance

Next, we would like to investigate how effective MoVR is in addressing the blockage problem. We place the AP on one side of the room and the mirror on an adjacent side, as shown in Fig. 2.[5] We place the headset at a random location and orientation. The AP transmits packets of OFDM symbols and the headset receives these packets and computes the SNR. We perform the experiment for 20 runs, changing the location and orientation of the headset for each run. We repeat each run for three scenarios:

[3] Although these systems use different technologies to track the user and headset, they all provide the location and position of the headset very accurately.

[4] Our test environment is the same as what HTC VIVE recommend for operation. For safety reason, they also suggest to move the furniture outside of the game area [11].

[5] Note, MoVR does not require the user to place the mirror carefully in a specific location, since its calibration mechanism is able to automatically localize the mirror.

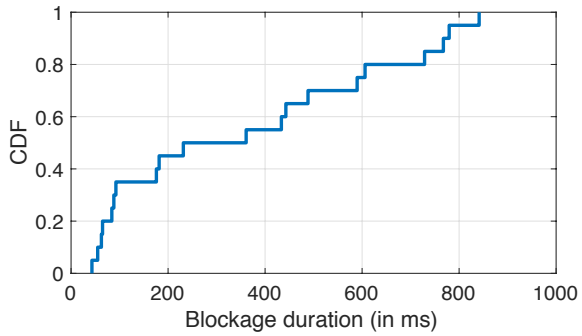


Figure 12—Blockage duration: The figure shows the CDF of the duration of line-of-sight blockages which happened during a 5-minute VR

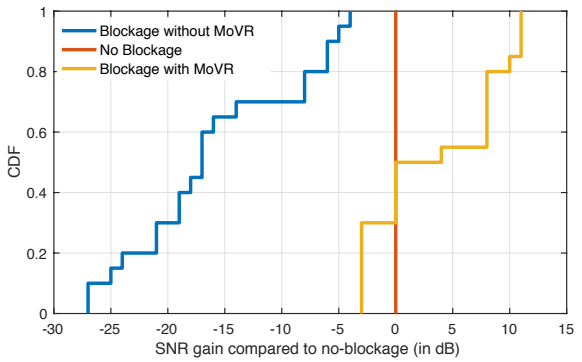


Figure 13—MoVR's mirror performance: Figure shows SNR gain compared to the *No-Blockage* in all three scenarios: *No-Blockage*, *Blockage-without-MoVR* and *Blockage-with-MoVR*.

- *No-Blockage:* In this scenario, there is a clear, direct path between the AP and the headset receiver. The AP and headset have aligned their beams along this path.
- *Blockage-without-MoVR:* In this scenario, the direct path from the AP is blocked. In the absence of a MoVR mirror, the best approach is to try to reflect the signal off of a wall or some other object in the environment. Thus, to find the best SNR possible without a mirror, we make the AP and the headset try all possible beam directions and pick the one that maximizes the SNR. The headset reports this maximum SNR.
- *Blockage-with-MoVR:* Here, we have the same blockage as in the previous scenario, but the system is allowed to use the MoVR mirror to reflect the signal as described in the earlier sections.

Fig. 13 compares the SNRs in all three scenarios. The figure plots the CDF of the SNR Gain relative to the SNR without blockage, defined as follows:

$$SNR\ Gain\ [dB] = SNR_{Scenario}[dB] - SNR_{No\ Blockage}[dB].$$

The figure shows that, in the absence of a MoVR mirror, a blockage drops the SNR by as much as 27dB, and the average SNR reduction is 17dB. As shown in §4, such high reduction in SNR prevents the link from supporting the required VR data rate. Thus, simply relying on indirect reflections in the environment to address blockage is ineffective.

The figure also shows that, for most cases, the SNR delivered using MoVR's mirror is higher than the SNR delivered over the direct line-of-sight path with no blockage. This is because, in those cases, the AP's distance to the mirror is shorter than its distance to the headset's receiver. Thus, the presence of MoVR's mirror along the path, and the fact that it amplifies the signal, counters the SNR reduction due to the longer distances to the headset. The figure further shows that, in some cases, MoVR performs 3dB worse than the no blockage scenario. This loss does not affect the data rate because, in these cases, the headset is very close to the AP, which provides a very high SNR (30dB) at the headset's receiver. This SNR is much higher than the 20dB needed for the maximum data rate. This experiment shows that MoVR's mirror enables a high data rate link between a VR headset and a PC even in the presence of blockage.

7.3 MoVR's Beam Alignment and Tracking Performance

As explained earlier, beam alignment is essential for mmWave links. In this section, we first investigate the accuracy of aligning the beam between the AP and the mirror. We then evaluate the accuracy of the beam alignment to the headset and the latency involved in establishing the alignment.

(a) Accuracy of beam alignment between the AP and the mirror: We evaluate MoVR's ability to find the best beam alignment between the AP and the mirror. We place the mirror somewhere in our testbed and estimate the angle which provides the best beam alignment between it and the AP, using the method described in §5.2. We repeat the experiment for 100 runs, changing the mirror location and orientation each time. We compare this to the ground truth angle, calculated from the locations of the AP and mirror. We use a Bosch GLM50 laser distance measurement tool to measure these locations to within a few millimeters.

Fig. 14 plots the angle estimated by MoVR versus the ground truth angle. The figure shows that MoVR estimates the angle of best beam alignment to within 2 degrees of the actual angle. Note that since the beam-width of our phased array is ∼10 degrees, such small error in estimating the angle results in a negligible loss in SNR.

(b) Beam alignment accuracy for the whole system: As explained in §5, MoVR leverages the location infor-

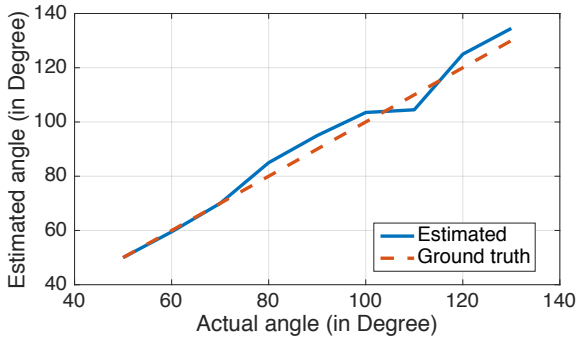


Figure 14—AP to mirror beam alignment accuracy: Angle estimated by MoVR (blue) versus the ground truth angle (red).

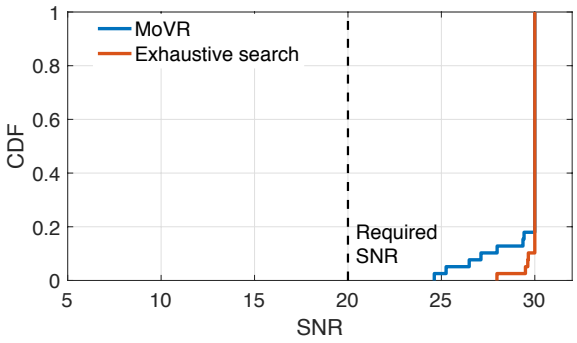


Figure 15—MoVR Beam Alignment Accuracy: The SNR of the signal at the VR receiver for two different scenarios: (1)MoVR's beam alignment algorithm and (2)exhaustive search. The figure shows that MoVR's beam alignment algorithm performs 4dB worse than exhaustive search in some cases. However, this loss does not affect the data rate, since the SNR is always much higher than the 20dB needed by the VR headset.

mation to track and align the transmitter's and receiver's beams. In this experiment, we evaluate the performance of MoVR in finding the best beam alignment to the headset (either from the AP or from the mirror). We place the headset at a random location and orientation in our testbed and compute the SNR it receives in two different scenarios: (1) MoVR's beam alignment algorithm and (2) Exhaustive search, which tries all possible combinations of AP, mirror and headset beam directions and picks the set which provide the highest SNR. We repeat this experiment 40 times, changing the location and orientation of the headset each time. Fig. 15 plots the results of this experiment. For most cases, MoVR's algorithm performs as well as Exhaustive search in finding the best beam alignment. MoVR's beam alignment performs 4dB worse than Exhaustive search in some cases, but this loss does not affect the data rate since the SNR is always much higher than the 20dB needed by the VR headset.[6] This result is significant since Exhaustive search requires trying all possible beam alignments, and hence introduces much latency and overhead. In contrast, given an initial calibration, MoVR obtains its alignment for free by leveraging the location information already available to the VR system.

(c) Beam alignment latency for the whole system: Next, we evaluate the capability of MoVR to perform beam alignment and tracking in real time. MoVR's beam alignment process includes multiple sources of delay, which sum to the total latency of the system. First, it takes $1ms$ for the VR tracking to update the headset position[14]. Our beam alignment algorithm, implemented in C++, uses this position to calculate new beam angles in $0.9\mu s$. Finally, MoVR's hardware (including the DACs and phase shifters) takes $1.7\mu s$ to reconfigure the beam [4, 5]. Given that our computation and the hardware reaction time are on the order of a few microseconds, the total delay is dominated by the VR location tracking delay, which is $1ms$. This delay is intrinsic to the VR system and is low enough to support the VR frame rate.

7.4 MoVR System Performance

Finally, we would like to evaluate the system as a whole and its ability to deliver the desired performance as the player moves around anywhere in the room. We place the headset at a random location and orientation in our VR testbed, and block the direct path between it and the AP with a hand. We then compute the SNR that the headset receives for three different scenarios: (1) No mirror, which tries all possible combinations of AP and headset beam directions and picks the one which provides the highest SNR; (2) Fixed gain mirror, where there is a mirror with a fixed amplification gain in our setup; and (3) MoVR, where we use our tracking algorithm and a mirror with our automatic gain control algorithm. We repeat this experiment 40 times, changing the location and orientation of the headset each time.

Fig. 16 plots the results of this experiment. The figure shows the received SNR at the headset for different room locations and for each scenario. Our results show that, in the presence of blockage, having a mirror with a fixed gain improves the SNR over relying on indirect reflections from the environment. However, there are still some locations with SNR below the 20dB needed by the VR headset. Adapting the mirror's amplifier gain improves the performance further and allows the system to achieve high SNR (24dB or higher) in all locations. This experiment confirms the need for our automatic gain control algorithm and shows that MoVR enables a high-quality untethered virtual reality, providing the required SNR for every location in a representative VR testbed.

[6] The maximum SNR is limited at 30dB because of the dynamic range of the hardware.

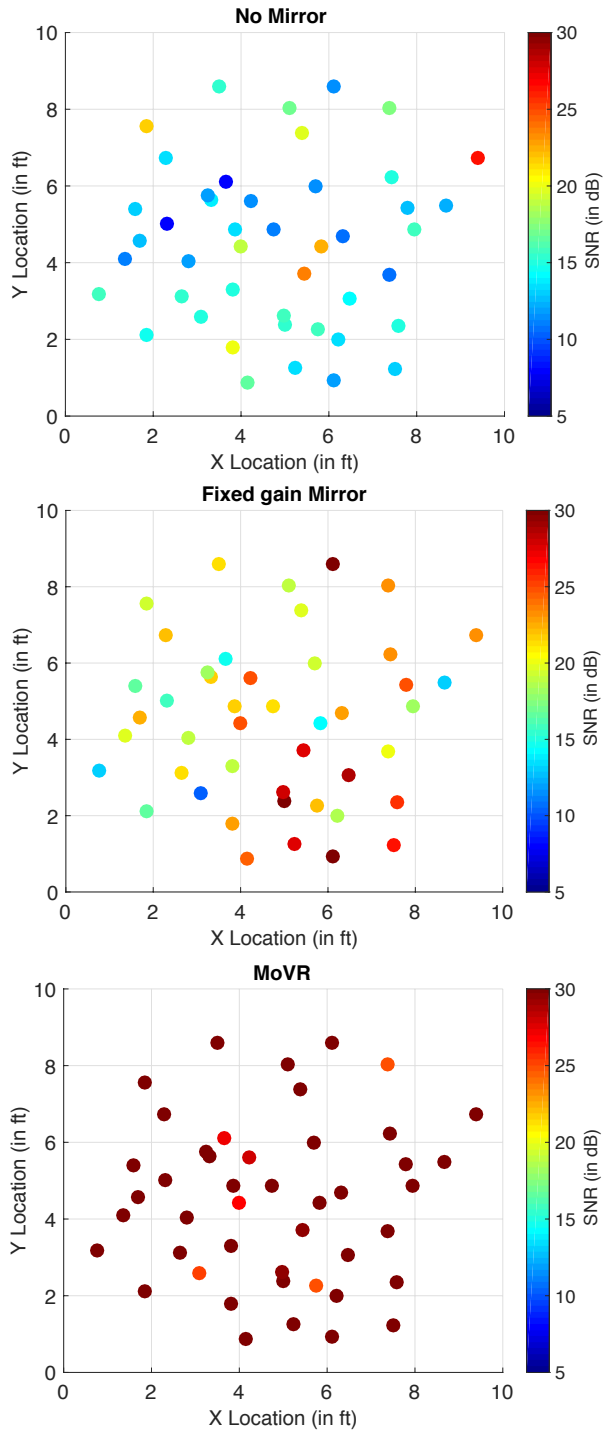


Figure 16—SNR at the headset's receiver The figure shows the SNR at the headset's receiver in the case of blockage for three different scenarios. (1) no mirror, (2) Fixed gain mirror, where there is a mirror with a fixed amplification gain in our setup, and (3) MoVR, where we use our beam tracking algorithm and a mirror with automatic gain control. The figure shows that during blockage the SNR is below the 20dB SNR needed by the VR headset for most locations. In contrast, MoVR enables high SNR in all locations.

8 Concluding Remarks

This paper presents MoVR, a system that enables a reliable and high-quality untethered VR experience via mmWave links. It provides a sustainable, high-data-rate wireless link to the VR headset even in the presence of blockage and mobility. In particular, it overcomes blockage of the mmWave link by introducing a smart and simple mmWave mirror that can reconfigure itself and adapt its angles of incidence and reflection. Further, MoVR introduces a novel algorithm that combines VR headset tracking information with RF-based localization to quickly steer the mmWave radios' beams and keep them aligned as the player moves around. Finally, it is worth mentioning that we have focused on eliminating the high-rate HDMI connection between the PC and headset. However, the current headset also uses a USB cable to deliver power. This cable can be eliminated by using a small rechargeable battery. The maximum current drawn by the mmWave radio and the HTC Vive headset is 1500mAh. Hence, a small battery (3.8x1.7x0.9in) with 5200mA capacity can run the headset for 3-4 hours [2]. An end-to-end evaluation of full system while VR data is streamed in real time and also improving the efficiency of mmWave hardware to increase the battery life are interesting avenues for future work.

Acknowledgments: We thank the NETMIT group, the reviewers and our shepherd, Heather Zheng for their insightful comments. This research is supported by NSF and HKUST. We thank the members of the MIT Center for their interest and support.

References

[1] *60 GHz: Taking the VR Experience to the Next Level.* http://www.sibeam.com/en/Blogs/2016/March/60GHZTakingtheVRExperience.aspx.

[2] *Anker Astro 5200mAh battery.* https://www.amazon.com/Anker-bar-Sized-Portable-High-Speed-Technology/dp/B00P7N0320.

[3] *CNET review for Samsung Gear VR.* http://www.cnet.com/products/samsung-gear-vr/.

[4] *Datasheet DAC-7228.* http://www.analog.com/media/en/technical-documentation/data-sheets/AD7228.pdf.

[5] *Datasheet Phase Shifter HMC933.* http://http://www.analog.com/media/en/technical-documentation/data-sheets/hmc933.pdf.

[6] *Facebook Expects to Ship 2.6 Million Oculus Rifts by 2017.* http://www.businessinsider.com/facebook-expects-to-ship-26-million-oculus-rifts-by-2017-2016-4.

[7] *FCC to explore 5G services above 24 GHz.* http://www.fiercewireless.com/tech/fcc-to-explore-5g-

services-auctioned-or-unlicensed-above-24-ghz.

[8] *HTC sold 15,000 $800 Vive virtual reality headsets in 10 minutes*. http://venturebeat.com/2016/02/29/htc-sold-15000-800-vive-virtual-reality-headsets-in-10-minutes/.

[9] *HTC sold 15,000 $800 Vive virtual reality headsets in 10 minutes*. http://www.ubeeinteractive.com/sites/default/files/Understanding%20Technology%20Options%20%20for%20Deploying%20Wi-Fi%20White%20Paper.pdf.

[10] *HTC Vive Oculus Rift Spec Comparison*. http://www.digitaltrends.com/virtual-reality/oculus-rift-vs-htc-vive/.

[11] *HTC VIVE Recommended Area*. http://www.ibtimes.co.uk/htc-vive-vr-how-much-room-space-do-i-really-need-1558494.

[12] *IEEE 802.16 Broadband Wireless Access Working Group*. http://ieee802.org/16/maint/contrib/C80216maint-05_112r8.pdf.

[13] *IEEE 802.16 Broadband Wireless Access Working Group*. http://www.keysight.com/upload/cmc_upload/All/22May2014Webcast.pdf?&cc=US&lc=eng.

[14] *Look Inside the HTC Vive's Positional Tracking System*. http://http://www.gamasutra.com/view/news/273553/An_expert_look_inside_the_HTC_Vives_positional_tracking_system.php.

[15] *mmWave 24GHz Transceivers*. http://www.infineon.com/cms/en/product/rf-and-wireless-control/mm-wave-mmic/channel.html?channel=db3a304339d29c450139d8bdb700579d.

[16] *Optoma's wireless VR headset frees you from PC cables*. http://www.pcworld.com/article/3044542/virtual-reality/optomas-new-wireless-vr-headset-frees-you-from-pc-cables.html.

[17] *Samsung Gear VR*. http://www.samsung.com/us/explore/gear-vr/.

[18] *Sulon sneak peak*. http://sulon.com/blog/sulon-q-sneak-peek.

[19] *Virtual Apple Airport Express*. http://www.apple.com/airport-express/.

[20] *Virtual Reality in Entertainment*. http://www.vrs.org.uk/virtual-reality-applications/entertainment.html.

[21] *Virtual Reality in Healthcare*. http://www.vrs.org.uk/virtual-reality-healthcare/.

[22] *Visus VR*. http://www.visusvr.com/.

[23] *VR for everyone*. https://vr.google.com/.

[24] *Wireless HDMI*. http://www.cnet.com/news/wireless-hd-video-is-here-so-why-do-we-still-use-hdmi-cables/.

[25] *WorldViz*. http://aecmag.com/technology-mainmenu-35/1130-news-worldviz-brings-warehouse-scale-vr-to-unreal-and-unity-engines.

[26] Wilocity 802.11ad Multi-Gigabit Wireless Chipset. http://wilocity.com, 2013.

[27] ABARI, O., BHARADIA, D., DUFFIELD, A., AND KATABI, D. Cutting the cord in virtual reality. In *HotNets* (2016).

[28] ABARI, O., HASSANIEH, H., RODREGUIZ, M., AND KATABI, D. Poster: A Millimeter Wave Software Defined Radio Platform with Phased Arrays. In *MOBICOM* (2016).

[29] BHARADIA, D., AND KATTI, S. Fastforward: Fast and constructive full duplex relays. In *SIGCOMM* (2014).

[30] BOYD, S. *Lecture 12: Feedback control systems: static analysis*. https://stanford.edu/~boyd/ee102/ctrl-static.pdf.

[31] C, C. S. *Advanced Techniques in RF Power Amplifier Design*. Artech House, 2002.

[32] CUI, Y., XIAO, S., WANG, X., YANG, Z., ZHU, C., LI, X., YANG, L., AND GE, N. Diamond: Nesting the Data Center Network with Wireless Rings in 3D Space. In *NSDI* (2016).

[33] DOYLE, J. C., FRANCIS, B. A., AND TANNENBAUM, A. R. *Feedback control theory*. Courier Corporation, 2013.

[34] ELTAYEB, M. E., ALKHATEEB, A., HEATH, R. W., AND AL-NAFFOURI, T. Y. Opportunistic Beam Training with Hybrid Analog/Digital Codebooks for mmWave Systems. In *GLOBESIP* (2015).

[35] FEDERAL COMMUNICATIONS COMMISSION. Title 47, code for federal regulations.

[36] GAO, B., XIAO, Z., ZHANG, C., JIN, D., AND ZENG, L. Joint SNR and Channel Estimation for 60 GHz Systems using Compressed Sensing. In *WCNC* (2013).

[37] GRAY, P. R., AND MEYER, R. G. Mos operational amplifier design-a tutorial overview. *IEEE Journal of Solid-State Circuits* (1982).

[38] HAIDER, M. K., AND KNIGHTLY, E. W. Mobility resilience and overhead constrained adaptation in directional 60 GHz WLANs: protocol design and system implementation. In *MobiHoc* (2016).

[39] HALPERIN, D., KANDULA, S., PADHYE, J., BAHL, P., AND WETHERALL, D. Augmenting Data Center Networks with Multi-Gigabit Wireless Links. In *SIGCOMM* (2011).

[40] HAN, S., I, C., XU, Z., AND ROWELL, C. Large-Scale Antenna Systems with Hybrid Analog and Digital Beamforming for Millimeter Wave 5G. *IEEE Communications Magazine* (January 2015).

[41] KIM, J., AND MOLISCH, A. F. Fast Millimeter-Wave Beam Training with Receive Beamforming. *Journal of Communications and Networks* (October 2014).

[42] RAMASAMY, D., VENKATESWARAN, S., AND MADHOW, U. Compressive tracking with 1000-element arrays: A framework for multi-gbps mm wave cellular downlinks. In *Allerton* (2012).

[43] RAPPAPORT, T. S., MURDOCK, J. N., AND GUTIERREZ, F. State of the art in 60GHz integrated circuits and systems for wireless communications. *Proceedings of the IEEE* (2011).

[44] SUR, S., VENKATESWARAN, V., ZHANG, X., AND RAMANATHAN, P. 60 GHz Indoor Networking through Flexible Beams: A Link-Level Profiling. In *SIGMETRICS* (2015).

[45] SUR, S., AND ZHANG, X. BeamScope: Scoping Environment for Robust 60 GHz Link Deployment. In *Information Theory and Application Workshop* (2016).

[46] SUR, S., ZHANG, X., RAMANATHAN, P., AND CHANDRA, R. BeamSpy: Enabling Robust 60 GHz Links Under Blockage. In *NSDI* (2016).

[47] YUAN, W., ARMOUR, S. M. D., AND DOUFEXI, A. An Efficient and Low-complexity Beam Training Technique for mmWave Communication. In *PIMRC* (2015).

[48] ZHOU, X., ZHANG, Z., ZHU, Y., LI, Y., KUMAR, S., VAHDAT, A., ZHAO, B. Y., AND ZHENG, H. Mirror on the Ceiling: Flexible Wireless Links for Data Centers. In *SIGCOMM* (2012).

[49] ZHU, Y., ZHANG, Z., MARZI, Z., NELSON, C., MADHOW, U., ZHAO, B. Y., AND ZHENG, H. Demystifying 60GHz Outdoor Picocells. In *MOBICOM* (2014).

Improving User Perceived Page Load Time Using Gaze

Conor Kelton*, Jihoon Ryoo*, Aruna Balasubramanian, and Samir R. Das
Stony Brook University
*student authors with equal contribution

Abstract

We take a fresh look at Web page load performance from the point of view of user experience. Our user study shows that perceptual performance, defined as *user-perceived page load time (uPLT)* poorly correlates with traditional page load time (PLT) metrics. However, most page load optimizations are designed to improve the traditional PLT metrics, rendering their impact on user experience uncertain. Instead, we present *WebGaze*, a system that specifically optimizes for the uPLT metric. The key insight in WebGaze is that user attention and interest can be captured using a user's eye gaze and can in turn be used to improve uPLT. We collect eye gaze data from 50 users across 45 Web pages and find that there is commonality in user attention across users. Specifically, users are drawn to certain regions on the page, that we call regions of high collective fixation. WebGaze prioritizes loading objects that exhibit a high degree of collective fixation to improve user-perceived latencies. We compare WebGaze with three alternate strategies, one of which is the state-of-the-art system that also uses prioritization to improve user experience. Our evaluation based on a user study shows that WebGaze improves median uPLT for 73% of the Web pages compared to all three alternate strategies.

1 Introduction

Web performance has long been crucial to the Internet ecosystem since a significant fraction of Internet content is consumed as Web pages. As a result, there has been a tremendous effort towards optimizing Web performance [21, 39, 60]. In fact, studies show that even a modest improvement in Web performance can have significant impact in terms of revenue and customer base [15, 16, 34].

The goal of our work is to improve page load performance, also called the Page Load Time (PLT), from the perspective of the user. PLT is typically measured using objective metrics such as *OnLoad* [11], and more recently *Speed Index* [32]. However, there is a growing concern that these objective metrics do not adequately capture the user experience [2, 5, 42, 49].

As a first step, we define a *perceptual* variation of page load time that we call *user-perceived PLT*, or *uPLT*. We conduct a systematic user study to show what was anecdotally known, i.e., uPLT does not correlates well with the OnLoad or Speed Index metrics. However, almost all current Web optimization techniques attempt to optimize for the OnLoad metric [10, 39, 47, 52, 62] rendering their impact on user experience uncertain. The problem is that improving uPLT is non-trivial since it requires information about user's attention and interest.

Our key intuition is to leverage recent advances in eye gaze tracking. It is well known that user *eye gaze* – in terms of fixation, dwell time, and search patterns – correlate well with user attention [17, 55]. In the human visual system only a tiny portion (about 2°) at the center of the visual field is perceived with the highest visual acuity and the acuity sharply falls off as we go away from the center [57]. Thus the eye must move when a user is viewing different parts of the screen. This makes eye gaze a good proxy for user's attention. Further, the commoditization of gaze trackers allow accurate tracking using low cost trackers [35, 37, 41, 51], without the need for custom imaging hardware.

We design *WebGaze*, a system that uses gaze tracking to significantly improve uPLT. WebGaze prioritizes objects on the Web page that are more visually interesting to the user as indicated by the user's gaze. In effect, WebGaze encodes the intuition that loading "important" objects sooner improves user experience. The design of WebGaze has two main challenges: *(i)* Scalability: personalizing the page load for each user according to their gaze does not scale to a large number of users, and *(ii)* Deployability: performing on-the-fly optimizations based on eye gaze is infeasible since page loads are short-lived and the gaze tracker hardware may not be available with every user.

WebGaze addresses these challenges by first distilling gaze similarities across users. Our gaze user study shows that most users are drawn to similar objects on a page. We divide the page into visually distinctive areas that we call regions and define the *collective fixation* of a region as the fraction of users who fixate their gaze on the region. Our study with 50 users across 45 Web pages shows that a small fraction of the Web page has extremely high collective fixation. For example, of the Web pages in our study, at least 20% of the regions were viewed by 90% of the users. Whereas, at least a quarter of the regions of the page are looked at by less than 30% of the users.

WebGaze then uses the HTTP/2 Server Push [13, 30] mechanism to prioritize loading objects on the page

that exhibit high degree of collective fixation. In fact, WebGaze provides a content-aware means of using the HTTP/2 Server Push mechanism. WebGaze does not require gaze tracking on-the-fly or require that every user participates in gaze tracking, as long as enough users participate to estimate the collective fixation. WebGaze's algorithm not only pushes the objects of interest, but also all dependent objects as obtained using the WProf tool [60].

The goal of WebGaze is to improve uPLT, a subjective metric that depends on real users. Therefore, to evaluate WebGaze, we conduct an extensive crowd-sourced user study to compare the performance of WebGaze's optimization with three alternatives: *Default*, *Push-All*, and *Klotski* [21]. Default refers to no prioritization. The Push-All strategy indiscriminately prioritizes all objects. Klotski is the state-of-the-art system whose goal is to improve Web user experience: Klotski works by prioritizing objects that can be delivered within the user's tolerance limit (5 seconds). We conduct user studies across 100 users each to compare WebGaze with each alternative.

The results show that WebGaze improves the median uPLT over the three alternatives for 73% of the 45 Web pages. In some cases, the improvement of WebGaze over the default is 64%. While the gains over the default case come from prioritizing objects in general, the gains over Push-All and Klotski come from prioritizing the right set of objects. *All user study data and videos of Web page loads under WebGaze and each alternative strategy can be found at* `http://gaze.cs.stonybrook.edu`.

2 Page Load Metrics

To study the perceptual performance of Web page loads, we define a perceptual variation of the PLT metric, that we call uPLT or user-perceived Page Load Time. uPLT is the time between the page request until the time the user 'perceives' that the page is loaded. In this section, we provide a background on traditional PLT metrics and qualitatively describe why they are different from uPLT. In the next section, we use a well posed user study to quantitatively compare traditional PLT metrics and uPLT.

OnLoad: PLT is typically estimated as the time between when the page is requested and when the OnLoad event is fired by the browser. The `OnLoad` event is fired when *all* objects on the page are loaded [8]. There is a growing understanding that measuring PLT using the OnLoad event is insufficient to capture user experience [2, 5, 49]. One reason is that users are often only interested in *Above-the-Fold (AFT)* content, but the OnLoad event is fired only when the entire page is loaded, even when parts of the page are not visible to the user. This leads to the OnLoad measure over-estimating the user-perceived latency. But in some cases, OnLoad can underestimate uPLT. For example, several Web pages load additional objects *after* the OnLoad event is fired. If the additional loads are critical to user experience, the PLT estimated based on the OnLoad event will underestimate uPLT. Variants of the OnLoad metric such as `DOMContentLoaded` [8, 60], are similarly disjoint from user experience.

Speed Index: Recently, `Speed Index` [32] was proposed as an alternate PLT measure to better capture user experience. Speed Index is defined as the average time for all AFT content to appear on the screen. It is estimated by first calculating the visual completeness of a page, defined as the pixel distance between the current frame and the "last" frame of the Web page. The last frame is when the Web page content no longer changes. Speed Index is the weighted average of visual completeness over time. The Speed Index value is lower (and better) if the browser shows more visual content earlier.

The problem with the visual completeness measure (and therefore Speed Index) is that it does not take into account the relative importance of the content. This leads to over- or under-estimation of user-perceived latency. If during the page load, a majority of visual components are loaded quickly, Speed Index estimates the page load time to be a small value. However, if the component critical to the user has not yet been loaded, the user will not perceive the page to be loaded. In other cases, Speed Index overestimates. For example, if a large portion of the page has visual content that is not interesting to the user, Speed Index will take into account the time for loading all the visual content, even though the user may perceive the page to be loaded much earlier.

Motivating Example: Figure 1 shows the `energystar.gov` page, and the three snapshots taken when the page was considered to be loaded according to the Speed Index, uPLT, and OnLoad metrics. In the case of uPLT, we choose the median uPLT value across 100 users who gave feedback on their perceived page load time (§3).

Speed Index considers the page to be loaded much earlier, at 3.2 seconds, even though the banner image is not loaded. For the users, the page is not perceived to be completely loaded unless the banner is loaded, leading to Speed Index under-estimating uPLT. On the other hand, the OnLoad metric estimates the page to be loaded 4 seconds *after* the user perceives the page to be loaded, even though their snapshots are the same visually. This is because the OnLoad event fires only when the entire page, including the non-visible parts, are loaded. This illustrative example shows one case when the traditional PLT metrics do not accurately capture user experience.

(a) Speed Index = 3.7 seconds (b) Median uPLT = 8.2 seconds (c) OnLoad = 12 seconds

Figure 1: Snapshots of the page load of `energystar.gov` shown at the Speed Index, the median uPLT across 100 users, and OnLoad values.

3 uPLT User Study

We conduct a user study to systematically compare uPLT with traditional PLT metrics, with the goal of verifying our observations presented in §2.

3.1 Set Up

Our user study was conducted (1) in the lab, and (2) online using crowd-sourcing. For the lab-based study we recruit subjects from our university. The user subjects belong to the age group of 25 to 40, both male and female. The online study is conducted on the Microworkers [9] platform. We present results from 100 users, 50 from each study. *All user studies presented in this paper were approved by the Institutional Review Board of our institution.*

3.2 User Study Set Up and Task

A key challenge of conducting Web page user studies *in-the-wild* is that the Web page load timings experience high variance [61]. The uPLT feedback from two users for a given page may not be comparable under such high variance. To conduct repeatable experiments we capture *videos* of the page load process. The videos are captured via `ffmpeg` at 10 fps with 1920x1080 resolution as the page loads. The users see the video instead of experiencing an actual page load on their computers. This way, each user views exactly the same page load process.

The primary task of the user is to report their perceived page load time when they are browsing the page. We ask the user to view the Web page loading process and give feedback (by pressing a key on the keyboard) when they perceive that the page is loaded. There is an inevitable reaction time between when a user perceives the page to be loaded and when they enter the key. For all measurements, we correct for the user's reaction time using calibration techniques commonly used in user-interaction studies [6]. To ensure high quality data from the user study, we remove abnormally early or late responses. To do so we utilize the `First Paint` and `Last Visual Change` PLT metrics [31]. The First Paint is the time between when the URL begins to load and the first pixel is rendered, and the Last Visual Change is the time when the final pixel changes on the user's screen. Any responses before the First Paint and after the Last Visual Change events are rejected.

3.2.1 Web Pages

In the default case, we choose 45 Web pages from 15 of the 17 categories of Alexa [3], ignoring Adult pages and pages in a language other than English. From each category, we *randomly* choose three Web pages; one from Alexa ranking 1–1000, another from Alexa ranking 10k–20k, and the other from Alexa ranking 30k+. This selection provides wide diversity in the Web pages. The network is standardized to the accepted DSL conditions [63], 50ms RTT, 1.3Mbps downlink and 384Kbps uplink, using the Linux traffic controller '`tc`' [18].

We conduct additional user studies by varying network conditions using the `tc` tool [18] to emulate: i) WiFi-like conditions: a 12 ms RTT link with 20 Mbps download bandwidth and ii) 3G-like conditions: a 150 ms RTT link with a 1.6 Mbps download bandwidth. We conduct these additional user studies across 30 users and 23 Web pages, half from the top 100 and remaining from between 10k–20k Web pages from Alexa's list [3].

3.2.2 Measurement Methodology

We load the Web page using Google Chrome version 52.0.2743.116 for all loads. We do not change the Web load process, and all the objects, including dynamic objects and ads, are loaded without any changes.

When the Web page load finishes, we query Chrome's Navigation Timeline API remotely through its Remote Debugging Protocol [22]. Similar interfaces exist on most other modern browsers [38]. From the API we are able to obtain timing information including the OnLoad measure. To estimate Speed Index, we first record the videos of the pages loading, recorded at 10 frames-per-second. The videos are fed through the WebPageTest Tool [63] that calculates the Speed Index.

3.3 Comparing uPLT with OnLoad and Speed Index

First, we compare the uPLT variations across lab-based and crowd-sourced studies for the same set of Web pages. Figure 2 shows the uPLT box plots for each Web page across the two different studies. Visually from the plot, we find that the lab and crowd-sourced users report similar distributions of uPLT. The standard deviation of the median uPLT difference between the lab and the crowd-sourced study for the same Web page is small, about 1.1 seconds. This same measure across Web pages is much larger, at about 4.5 seconds. This increases our

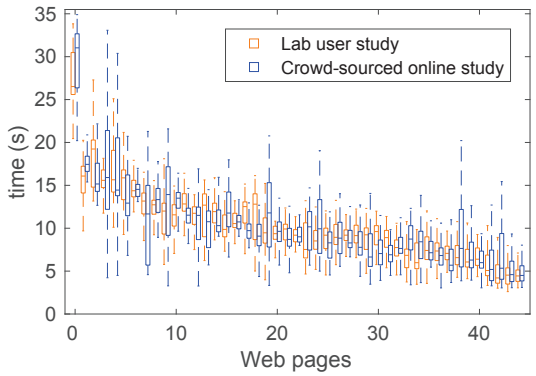


Figure 2: Comparing the uPLT box plot between the 50 lab and 50 crowd-sourced users. Although the uPLT values vary across users, the distributions are similar for the two data sets.

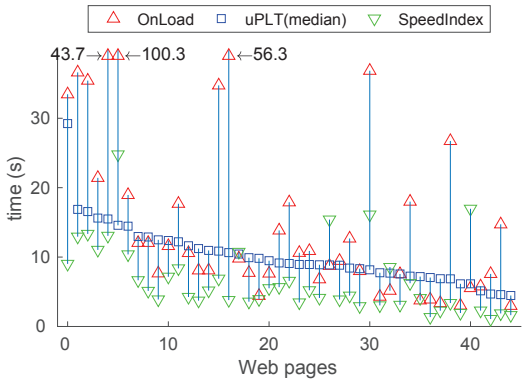


Figure 3: Comparing median uPLT with OnLoad and Speed Index across 45 Web pages and 100 users. The median uPLT is lower than OnLoad for 50% of the Webpages, and higher than Speed Index for 87% of Webpages.

confidence in the results from the crowd-sourced user study; we leverage a similar crowd-sourced study to evaluate WebGaze.

Figure 3 shows median uPLT compared to the OnLoad and Speed Index metrics across the 45 pages and 100 users, combining the crowd-sourced online study and the lab study. The Speed Index and OnLoad values are calculated from the same Web page load in which was recorded and shown to the users.

We observe that uPLT is not well correlated with the Onload and Speed Index metrics: the Correlation Coefficient between median uPLT and the OnLoad metric is ≈ 0.46 while the correlation between median uPLT and the Speed Index is ≈ 0.44. We also find the correlation between uPLT and the DomContentLoaded to be ≈ 0.21.

The OnLoad metric is about 6.4 seconds higher than the median uPLT on an average, for close to 50% of the pages. For 50% of Web pages, the OnLoad is lower than the median uPLT by an average of about 2.1 seconds. On the other hand, Speed Index, estimated over visible AFT content, is about 3.5 seconds lower than uPLT for over 87% of the Web pages. In Section 2 we discussed the cases in which the OnLoad and Speed Index can over and underestimate the user perceived page loads. From our results we see that while cases of uPLT over and underestimation occur in equal proportion for the OnLoad, the case of uPLT underestimation, as shown in Figure 1, occurs more for the Speed Index.

3.4 uPLT Across Categories

The 45 Webpages used in the study have diverse characteristics. In Figure 4, we study how uPLT differs from traditional PLT metrics for different categories. Each point in the plot is the median uPLT across 100 users.

We divide the Web pages across four (4) categories: *(i)* Light html: landing pages such as `google.com`, *(ii)* CSS-heavy; *(iii)* Javascript-heavy; and *(iv)* Image-heavy. To categorize the page into the latter three categories, we look at the types of objects downloaded for each page and count the number of CSS, Javascript, and images. The categories are based on the type of object that is fetched most when the page is loaded.

Light html and *CSS-heavy* pages are simple and see little difference between the uPLT and the OnLoad and Speed Index metrics. However, for pages with a lot of dynamic Javascript, the median difference between uPLT and OnLoad is 9.3 seconds. Similarly, for image-heavy pages, the difference between uPLT and OnLoad is high. This is largely because, as the number of images and dynamic content increases, the order in which the objects are rendered becomes important. As we show in the next section, users typically only focus on certain regions of the page and find other regions unimportant, making it critical that the important objects are loaded first.

3.5 Varying Network Conditions

Finally, to verify the robustness of our results, we analyze the differences between uPLT OnLoad, and Speed Index under varying network conditions.

Under the slower 3G-like network conditions across 30 lab users and 23 Web pages, median uPLT poorly correlates with OnLoad and Speed Index with a correlation coefficient of 0.55 and 0.51 respectively. The median uPLT was greater than Onload 46% of times, with a median difference of 4.7 seconds. The uPLT was less than Speed Index 72% of the time with the median difference of 1.86 seconds. When we evaluate under WiFi-Like conditions we find the correlation between between

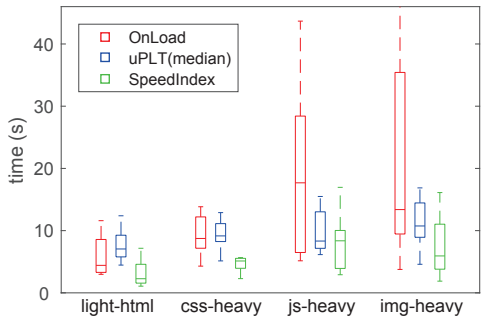


Figure 4: OnLoad, SpeedIndex and uPLT for different categories of Web pages

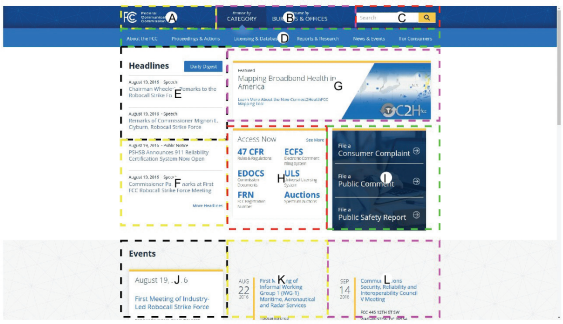


Figure 5: Segmentation of the Web page of `fcc.gov` into visual regions. The visual regions are named "A", "B", "C", etc.

OnLoad and uPLT is much higher at .74. This result is likely because in faster networks, more pages load instantaneously causing the user perceived latency to not differ much from the OnLoad.

4 Gaze Tracking

Existing Web page optimizations focus on improving traditional PLT metrics. However, our analysis shows that traditional PLT metrics do not correlate well with uPLT, rendering the effect of existing optimizations on user experience unclear. Instead, we propose to leverage users' eye gaze to explicitly improve uPLT.

4.1 Inferring User Interest Using Gaze

Gaze tracking has been widely used in many disciplines such as cognitive science and computer vision to understand visual attention [23, 40]. Recently, advances in computer vision and machine learning have also enabled low cost gaze tracking [35,37,41,51]. The low cost trackers do not require custom hardware and take into account facial features, user movements, a user's distance from the screen, and other user differences.

WebGaze leverages the low cost gaze trackers to capture visual attention of users. As a first step, we conduct a user study to collect eye gaze from a large number of users across Web pages. Using gaze data collected using both a low cost gaze tracker and an expensive custom gaze tracker, we show that the tracking accuracy of the low cost tracker is sufficient for our task.

Next, we analyze the collected gaze data to infer user patterns when viewing the same Web page. Specifically, we identify the *collective fixation* of a region on the Web page, which presents a measure to represent how much a broad group of users attention is fixated on the specific region. WebGaze uses collective fixation as a proxy for user interest, and leverages it to improve uPLT.

4.2 Gaze User Study Set Up

The gaze user study set up is similar to the lab user study described in §3.1. Recall that in our lab user study, we collect uPLT feedback from 50 users as they browse 45 Web pages. In addition to obtaining the uPLT feedback, we also also capture the user's eye gaze.

The gaze tracking is done using an off-the-shelf webcam-based software gaze tracker called GazePointer [27]. GazePointer tracks gaze at 30 frames/sec and does not require significant computational load because it uses simple linear regression and filtering techniques [37, 56] unlike gaze trackers that require more complicated machine learning [25]. We use a 1920 x 1080 resolution 24 inch monitor with a webcam mounted on top. The screen is placed at a reading distance ($\approx$ 50cm) from the participant. We perform a random point test, where we ask the users to look at 100 predetermined points on the screen. We find the error of tracker to be less than 5° at the 95th percentile.

The user study requires gaze calibration for each user; we perform this calibration multiple times during the study to account for users shifting positions, shifting directions, and other changes. We can potentially replace this calibration requirement using recent advances in gaze tracking that utilize mouse clicks for calibration [41]. These recent calibration techniques are based on the assumption that the user's gaze will follow their mouse clicks which can then be used as ground truth for calibration.

We augment the gaze study with an auxiliary study using a custom gaze tracker with 23 users. The study set up is similar to above, except we use a state-of-the-art *Eye Tracking Glasses 2 Wireless* gaze tracker manufactured by SMI [48]. The gaze tracker uses a custom eyeglass, tracks gaze at 120 frames/sec, and has a very high accuracy ($\approx 0.5°$ is typical).

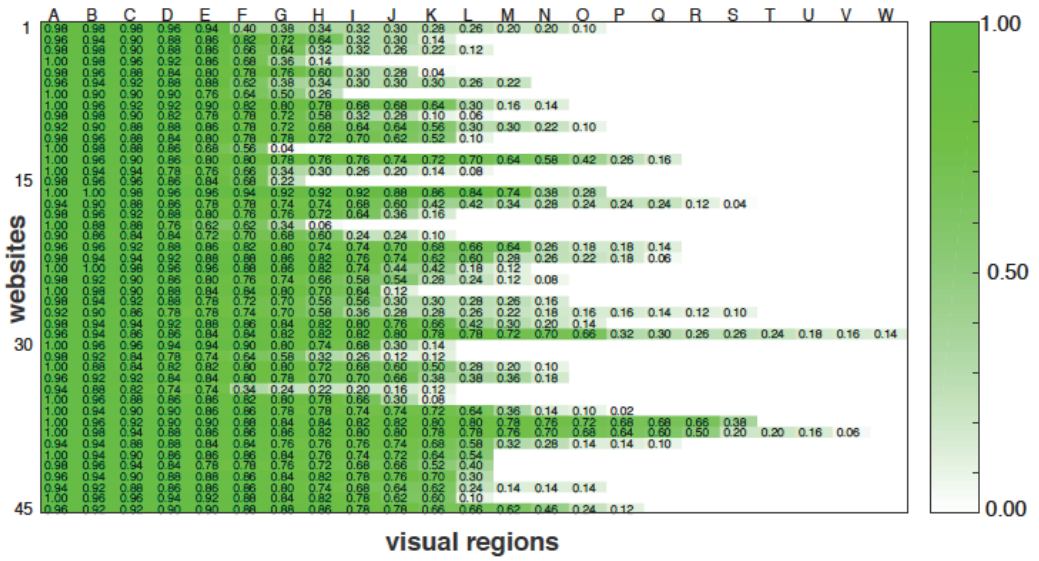


Figure 6: A heatmap of the collective fixation of Web page visual regions. Rows correspond to Web pages and the columns correspond to visual regions. For example, for Web site 1, visual region "A" has a collective fixation of 0.98 which means 98% of the users fixated on region "A" during gaze tracking.

4.3 Gaze Tracking Methodology

When a human views a visual medium, his/her eyes exhibit quick jerky movements known as *saccades* interspersed with relatively long ($\approx$ .5 second) stops known as *fixations* that define the his/her interest [33].

Web pages are designed for visual interaction, and thus contain many visually distinct elements, or *visual regions* [12], such as headers, footers, and main content sections, that help guide a user when viewing the page. Rather than track each fixation point, we segment a Web pages into its set of visual regions and track only the regions associated with the user's fixation points [24]. Figure 5 shows an example segmentation of `fcc.gov` into its visual regions. It is from this representation that we estimate the collective fixation of a visual region as the fraction of all users' gaze tracks that contain a fixation on the visual region. As part of future work, we will explore other signals of a user's gaze, including fixation duration and fixation order.

4.4 Collective Fixation Results

Figure 6 shows the collective fixation across each visual region of each Web page. The rows correspond to the Web page and the columns correspond to the visual regions in the Web page labeled 'A', 'B', etc (see example in Figure 5). Note that different Web pages may have different visual regions, since region creation depends on the overall page structure.

Figure 6 shows that for the first Web page, 5 regions have a collective fixation of over 0.9. In other words, 90% of the users fixated on these 5 regions in gaze tracking. But the remaining 75% of the regions have a collective fixation of less than 0.5.

In general, we find that across the Web pages, at least 20% of the regions have a collective fixation of 0.9 or more. We also find that on an average, 25% of the regions have a collective fixation of less than 0.3; i.e., 25% of the regions are viewed by less than 30% of the users.

Figure 7 shows the data in Figure 6 from a different perspective. Figure 7 is the median of the CCDF's of collective fixations for each site. Each point in the graph shows the percentage of regions with at least a certain collective fixation value. For example, the graph shows that 55% of the regions have a collective fixation of at least 0.7 in the median case. Our key takeaways are: *(i)* several regions have high collective fixation, and *(ii)* there is a significant number of regions that are relatively unimportant to the users. These points suggest that a subset of regions are relevant to the users' interests, an observation that can be exploited to improve uPLT (§6).

Figure 8 shows a visualization of the gaze tracks on `fcc.gov` across all users. The combined gaze fixations show a high degree of gaze overlap. The thicker lines show the regions on the Web page where the users' gaze exhibit a high degree of collective fixation. The thinner lines show the regions that only a few users look at.

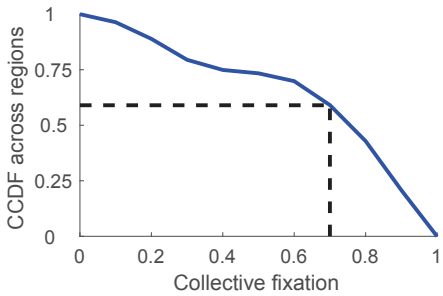


Figure 7: The median of the CCDF's of collective fixations across regions. Each point in the graph shows the fraction of regions with at least a certain collective fixation value in the median case.

Figure 8: A visualization of the gaze of all users when viewing `fcc.gov`. Certain regions on the page have more gaze fixations than others (as evidenced by the thicker lines).

4.5 Auxiliary Studies

In our auxiliary studies, we track gaze using a state-of-the-art gaze tracker as users viewed Web page loads under slow 3G and fast WiFi-like network conditions (network set up discussed in §3.1). The collective fixation results using the custom gaze tracker are quantitatively similar to the results when tracking gaze using the low cost tracker. For instance, 30% of the regions have a collective fixation of more than 0.8, and 30% of regions have a collective fixation of less than 0.1 under slow network conditions. The results under fast network conditions are similar.

We also conducted an additional set of experiments to study the effects personalized Web pages have on the user's gaze. Web pages such as `Facebook` customize their page to a given user, even though the overall structure of the page remains the same. This customization may result in different users focusing on different parts of the page. We choose five personalized Web pages where the users login to the site: Facebook, Amazon, YouTube, NYTimes, CNN. We conduct a user study with 20 users who gave us permission to track their gaze while they were browsing the logged-in Web pages. Despite customized content, we see similar patterns in collective fixation. All sites see a collective fixation of 0.8 or above for 30% of regions while still having at least 30% of regions with collective fixations below 0.1. In addition, on average these sites have 20% of their regions with a collective fixation above 0.9 and 33% below 0.3. Thus, even for pages where specific contents of the page vary across users, we observe there exist regions of high and low collective fixation.

5 WebGaze Design and Architecture

The previous section establishes that for each Web page, there exists several regions with high collective fixation. WebGaze is based on the intuition that prioritizing the loading of these regions can improve uPLT. This intuition is derived from existing systems and metrics, including Klotski [21] and the Speed Index. The goal of the Klotski system is to maximize the number of objected rendered within 3–5 seconds, with the intuition that loading more objects earlier improves user experience. Similarly, Speed Index uses the visual loading progress of a page as a proxy for the user's perception. The Speed Index value improves when more objects are rendered earlier on the screen. Similar to these works, our goal is also to render more objects earlier, but WebGaze chooses objects that are more important to the users as determined by their gaze feedback.

5.1 Architecture

Figure 9 shows the architecture of WebGaze. WebGaze is designed: *(i)* to have no expectations that all users will provide gaze feedback, *(ii)* to not require that pages be optimized based on real time gaze feedback. We note that existing gaze approaches for video optimization do rely on real time gaze feedback for prioritization [43]. However, Web page loads are transient; the short time scales makes it infeasible to optimize the Web page based on real time gaze feedback.

The WebGaze architecture collects gaze feedback from a subset of users as they perform the browsing task. WebGaze collects the gaze feedback at the granularity of visual regions. When sufficient gaze feedback is collected, the WebGaze server estimates the collective fixation across regions. The server periodically collects more gaze information and updates its fixation estimations as the nature of the Web page and users' interests change.

Based on the collective fixation values, WebGaze, (1) identifies the objects in regions of high collective fixation, (2) extracts the dependencies for the identified objects, (3) uses HTTP/2 Server Push to prioritize the identified objects along with the objects that depend on them. Below, we describe these steps in detail.

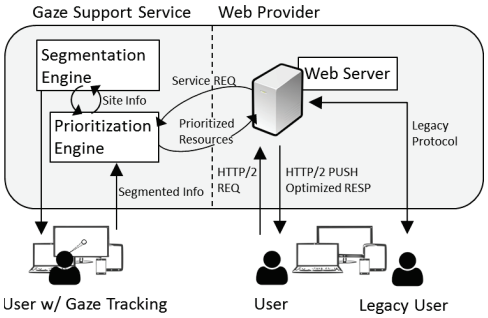


Figure 9: WebGaze architecture.

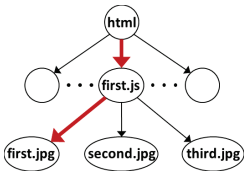


Figure 10: A dependency graph for an example page. If the `first.jpg` needs to be prioritized based on the collective fixation measure, then `first.js` also needs to be prioritized since `first.jpg` depends on it.

5.2 Identifying Objects to Prioritize

To identify which Web objects to prioritize, we use a simple heuristic: if a region has a collective fixation of over a *prioritization threshold*, then the objects in the region will be prioritized. In our evaluation, we set the prioritization threshold to be 0.7, thus any objects within a visual region that has a collective fixation of 0.7 or higher are prioritized. Recall from Figure 7 that this value identifies 55% of regions as candidates for prioritization in the median case.

Moving this threshold in either direction incurs different trade-offs. When the prioritization threshold is increased (moving right in Figure 7) we become more conservative in identifying objects to prioritize. However, in being more conservative we may miss prioritizing regions of which are important to some significant minority of users, which can in-turn negatively affect the aggregate uPLT. When the prioritization threshold is decreased, more regions are prioritized. The problem is that prioritizing too many objects leads to data contention for bandwidth that in turn affects uPLT [14] (in §6 we show the effect of prioritizing too many objects.) Empirically, we find that the prioritization threshold we chose works well in most cases (§6), through it can be further tuned.

Since each region may have multiple objects, WebGaze extracts the objects that are embedded within a region. To do this, we query the Document Object Model (DOM) [4], which is an intermediate representation of the Web page created by the browser. From the DOM we obtain the CSS bounding rectangles for all objects visible in the 1920x1080 viewport. An object is said to be in a given region if its bounding rectangle is within the region. If an object is said to belong to multiple regions, we assign the maximum of the collective fixation of the regions to the object.

5.3 Extracting Dependent Objects

Web page objects are often dependent on each other and these dependencies dictate the order in which the objects are processed. Figure 10 shows an example dependency graph. If `first.jpg` belongs to a region with high collective fixation and is considered for prioritization, then `first.js` also needs to be prioritized, because `first.jpg` depends on `first.js`. If not, then prioritizing `first.jpg` is not likely to be useful since the browser needs to fetch and process `first.js` before processing the image.

Our system identifies dependencies of each object to be prioritized, and considers these dependent objects for prioritization as well. Our current implementation uses WProf [60] to extract dependencies, but other dependency tools [21, 39] can also be used. While the contents of sites are dynamic, the dependency information has shown to be temporally stable [21,39]. Thus, dependencies can be gathered offline.

5.4 Server Push and Object Sizes

WebGaze, like other prioritization strategies [21], uses HTTP/2's Server Push functionality to implement the prioritization. Server Push decouples the traditional browser architecture in which Web objects are fetched in the order in which the browser parses the page. Instead, Server Push allows the server to preemptively push objects to the browser, even when the browser did not explicitly request these objects. Server Push helps *(i)* by avoiding a round trip required to fetch an object, *(ii)* by breaking dependencies between client side parsing and network fetching [39], and *(iii)* by better leveraging HTTP/2's multiplexing [14].

Of course, Server Push is still an experimental technique and is not without problems. Google's studies find that using Server Push can, in some cases, result in a reordering of critical resources that leads to pathologically long delays [14]. To avoid such pathological cases, we check for a simple condition: if the FirstPaint of the page loaded with WebGaze takes longer than the LastVisualChange in the default case, we revert back to the default case without optimization (recall the definitions of FirstPaint and LastVisualChange from §3.1). In our evaluation, we found that for 2 out of the 45 pages, WebGaze's use of Server Push resulted in such delays.

Another problem is that Server Push can incur performance penalties when used without restraint. Pushing too many objects splits the bandwidth among the objects, potentially delaying critical content, and in-turn, worsening performance. To address this, Klotski avoids prioritizing large objects or objects with large dependency chains [21]. Although we do not consider object sizes in our current implementation, we plan to do so as part of future work.

Finally, Server Push can use *exclusive* priorities [13] to further specify the order in which the prioritized objects are pushed as to respect page dependencies. However, existing HTTP/2 implementations do not support fully this feature. With a full implementation of HTTP/2's exclusive priorities, WebGaze's mechanism can potentially be tuned even further.

6 WebGaze Evaluation

We conduct user studies to evaluate WebGaze and compare its performance with three alternative techniques which are:

- *Default*: The page loads *as-is*, without prioritization
- *Push-All*: Push all the objects on the Web page using Server Push. This strategy helps us study the effect of Server Push at its extreme.
- *Klotski*: Uses Klotski's [21] algorithm to push objects. The algorithm pushes objects and dependencies with the objective of maximizing the amount of ATF content that can delivered within 5 seconds.

As before (§3.1), we record videos of the Web page as it is loaded using WebGaze and each alternate technique. The users provide feedback on when they perceive the page to be loaded as they watch the video. We conduct the user study across 100 users to compare WebGaze and each alternative technique. Videos of the Web page loads, under each technique, are available on our project Web page, `http://gaze.cs.stonybrook.edu`.

6.1 Methodology

Web pages: We evaluate over the same set of 45 Webpages as our uPLT and gaze studies (§3.1). Recall, from the WebGaze architecture, that the Web server corresponding to each Web page prioritizes content based on input from WebGaze. For evaluation purposes, we run our own Web server instead and download the contents of each site locally. We assume that all cross-origin content is available in one server. We note that HTTP/2 best practices suggest that sites should be designed such that as much content as possible is delivered from one server [14]. Nondeterministic dynamic content, such as ads, are still loaded from the remote servers.

Server and client: The Web pages are hosted on an Ubunbu 14.04 server running version 2.4.23 of Apache `httpd` which supports HTTP/2 protocol and Server Push functionality. The Web client is Chrome version 52.0.2743.116, which supports both the HTTP/2 protocol and Server Push, that is also run on an Ubuntu 14.04 machine. Traffic on the client machine is controlled using `tc` [18] to replicate standard DSL conditions (§3.1). When using push, we use default HTTP/2 priorities. Due to the standardized conditions of our network, the average variance in OnLoad is less than 1%. So we are able to compare the uPLT values across different page loads.

User study: We conduct pairwise comparisons of uPLT. To this end, we show the users randomized Web page loads that are loaded using WebGaze and using one of the three alternatives. The users provide feedback on uPLT. For each set of 45 comparisons, we recruit 100 users, for a total of 300 users. An alternative design would be to conduct a user study where a single user provides feedback for Web page loads under all four alternatives; but this requires users to give feedback on 180 Web pages which becomes tedious.

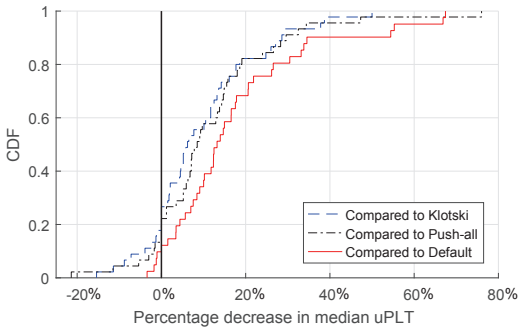


Figure 11: CDF of improvement in uPLT over Default, Push-All, and Klotski across the 100 users and 45 Web pages.

6.2 Comparing WebGaze with Alternatives

Figure 11 shows the CDF of the percentage improvement in uPLT compared to alternatives. On an average, WebGaze improves uPLT 17%, 12% and 9% over Default, Push-All, and Klotski respectively. At the 95th percentile, WebGaze improves uPLT by 64%, 44%, and 39% compared to Default, Push-All, and Klotski respectively. In terms of absolute improvement, when WebGaze improves uPLT the improvement is by an average of 2 seconds over Default and Push-All, and by an average of 1.5 seconds over Klotski. At the 90th percentile, WebGaze improves uPLT by over 4 seconds.

In about 10% of the cases WebGaze does worse than Default and Push All in terms of uPLT and in about 17%

Alternative	# WebGaze better	# WebGaze same	# WebGaze worse
Default	37	4	4
Push-All	35	4	6
Klotski	33	4	8

Table 1: Number of Web pages for which WebGaze performs better, same, and worse, in terms of uPLT in the median case compared to the alternatives.

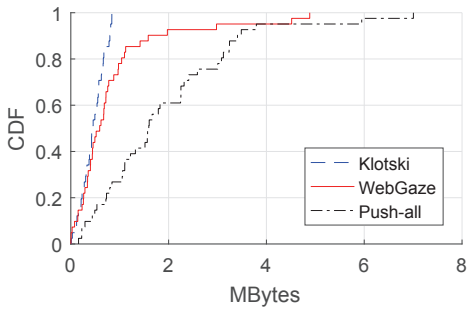


Figure 12: The total size of pushed objects under WebGaze, Klotski, and Push-All.

of the cases, WebGaze performs worse than Klotski. Of these cases where the competing strategies outperform WebGaze, the average reduction in performance is 13%.

Table 1 shows the number of Web pages for which WebGaze performs better, the same, and worse in terms of uPLT for the median case, as compared to the alternatives. Next we analyze the reasons for the observed performance differences.

6.3 Case Study: When WebGaze Performs Better

It is not surprising that WebGaze improves uPLT over the default case. Recall our intuition based on prior work [21, 32] that prioritizing regions with high collective fixation can improve uPLT. In addition, pushing objects with adherence to their dependencies has been shown to improve page load performance [39, 60].

Push-All is an extreme strategy, but it lets us study the possible negative effects of pushing too many objects. We find that Push-All delays critical object loads and users see long delays for even the First Paint [31]. In our study, Push-All increases First Paint by an average of 14% compared to WebGaze. Push-All, in-turn, tends to increase uPLT. The problem with pushing too many objects is that each object only gets a fraction of the available bandwidth, in spite of techniques such as HTTP/2 priorities [14].

Different from uPLT, for OnLoad, it is more critical that all objects are loaded even if objects critical to the user are delayed. We see this tendency in our results: the Push-All strategy in fact improves OnLoad for 11 of the 45 pages, whilst hurting uPLT. This example shows that optimizations can help OnLoad, but hurt uPLT.

The uPLT improvement compared to Klotski comes from content-aware prioritization. In the case of Klotski, ATF objects are pushed based on whether they will be delivered within 5 seconds. This may not correlate with the objects that the user is interested in. For example, the Webpage `www.nysparks.com`, Klotski prioritizes the logo.jpg image which is in a region of low collective fixation. This essentially delays other more critical resources that are observed by a large number of users.

6.4 Case Study: When WebGaze Performs Worse

WebGaze performs worse than Klotski in 17% of the cases with a median performance difference of 5.5% and a maximum difference of 15.4%. In each of these cases, we find that Klotski sends less data compared to WebGaze and is more conservative. Figure 12 shows the relative size of objects pushed by WebGaze and Klotski across the Web pages. This suggests that we need to perform more analysis on determining the right amount of data that can be pushed without affecting performance. Similarly, when compared to Default, WebGaze performs worse for 4 of the 45 Webpages. In each of these cases, WebGaze pushed too much data causing bandwidth contention.

In all cases when WebGaze performs worse compared to Push-All, we find that the Web pages were smaller, less than 1.2 MB. We speculate that pushing all objects for pages of small sizes does not force as much contention for bandwidth.

6.5 Case Study: When WebGaze Performs the Same

For a fraction of less than 10% of the pages we find that WebGaze performs similar to the other alternatives. For two of the Web pages, the uPLT values are similar across the four alternatives. In other words, Server Push did not change performance. This could be because the default page itself is well optimized. For the other two pages, WebGaze's Server Push resulted in pathologically delays, and therefore the pages were not optimized (§5.4).

Although they are not the metrics WebGaze intends to optimize, for completeness we briefly discuss the performance of WebGaze in terms of the OnLoad, Speed Index, and First Paint. In terms of all three metrics, WebGaze and Klotski show comparable performance. In comparison to Default and Push-All, WebGaze shows only 1–3% improvement in the OnLoad. WebGaze improves the

Speed Index metric by an average of 18% compared to the Push-All strategy. However, there is no difference in the average Speed Index measure between WebGaze and Default. Lastly, as discussed earlier, WebGaze improves the average First Paint metric by 14% compared to Push-All. However WebGaze does increase the time to First Paint by 19% on average compared to Default, thus improving uPLT despite increasing the First Paint overall. This result loops back to our intuition (§5) that loading more objects important to the user sooner is critical to uPLT.

7 Related Work

We discuss three related lines of research that are relevant to our work: Web performance, page load measurements, and modeling user attention for Web pages.

7.1 Improving Web Performance

Given the importance of Web performance, significant research effort has gone into improving Web page loads. These efforts include designing new network protocols [10, 52], new Web architectures [39, 47, 62], best practices [1], and tools to help developers write better Web pages [29]. However, most of these efforts target the traditional *OnLoad* metric.

More recently, systems such as Klotski [21] are targeting the user quality of experience rather than optimizing traditional PLT metrics. As discussed earlier, Klotski uses HTTP/2's Server Push functionality to push *high utility and visible* objects to the browser. WebGaze uses a similar prioritization technique, but prioritizes objects based on user interest. Our evaluations show that WebGaze improves uPLT compared to Klotski across 100 users (§6).

7.2 Page Load Measurements

The research community has reported on a broad spectrum of Web page user studies. On the one end, there are controlled user study experiments [53], where the researchers create specific tasks for the subjects to complete. However, to create repeatable user studies and to control the Web page load times, the authors create *fake* Web pages. On the other end, there are large scale, less controlled studies [20] that measure performance of hundreds of real Web pages. But these studies only measure objective metrics such as the OnLoad metric.

Around the same time as the design and development of WebGaze, researchers have developed a similar testbed called `eyeorg` to crowd-source Web user experience [54]. The eyeorg study also uses a user-perceived PLT metric to measure user experience, and records the Web pages to obtain standardized feedback from the users as to when they feel the page is loaded. Their methodology in obtaining feedback is slightly different from our study in that they allow the users to transition frame by frame before providing their uPLT. The eyeorg study finds high correlation between the OnLoad and uPLT metrics, similar to our findings in the WiFi-like environment. Different from the eyeorg study, we vary the network conditions when loading the page and show that the correlation results depend on the underlying network (§3). On slow networks, OnLoad and uPLT are poorly correlated, while in faster networks, OnLoad and uPLT are better correlated; the later corroborating more with the results of eyeorg. Going beyond crowd-sourcing uPLT feedback, our work also shows how uPLT can be improved by leveraging eye gaze.

7.3 Web Saliency

The computer vision community has widely studied how eye gaze data can be used as ground truth to build saliency models [28,59]. Saliency defines the features on the screen that attract more visual attention than others. Saliency models predict the user's fixation on different regions of the screen and can be used to capture user attention without requiring gaze data (beyond building the model). While most of the research in this space focuses on images [44, 64], researchers have also built saliency models for Web pages.

Buscher et al. [19] map the user's visual attention to the DOM elements on the page. Still and Masciocchi [50] build a saliency model and evaluate for the first ten (10) fixations by the user. Shen et al. [46] build a computational model to predict Web saliency using a multi-scale feature map, facial maps, and positional bias. Ersalan *et al.* [24] study the scan path when the user is browsing the Web. Others have looked at saliency models for Web search [45] and text [26, 58].

However, existing Web saliency techniques have relatively poor accuracy [19, 46]. This is because predicting fixations on Web pages is inherently different and more challenging compared to images: Web pages, unlike images, are a mix of text and figures. Web page loading is an iterative process where all objects are not rendered on the screen at the same time, and there is a strong prior when viewing familiar Web pages.

Our work is orthogonal to the research on Web saliency. WebGaze can leverage better Web saliency models to predict user interest. This will considerably reduce the amount of gaze data that needs to be collected, since it will only be used to provide ground truth. We believe that our findings on how gaze data can improve user-perceived page load times can potentially spur research on Web saliency.

8 Discussions

There are several technical issues that will need a close look before a gaze feedback-based Web optimization can be widely useful.

Mobile Devices: It is expected that more and more Web content will be consumed from mobiles. Mobile devices bring in two concerns. First, errors in gaze tracking may be exaggerated in mobiles as the screen could be too small, or the performance of gaze tracking on mobile could be too poor. Significant advances are being made on camera-based gaze tracking for mobile smartphone class devices [7]. But, accuracy is also as not critical to our approach as we require the gaze to be tracked at the granularity of large visual regions.

A second concern is that gaze tracking on mobile devices may consume additional amounts of energy [43]. This is due to the energy consumed in the imaging system and on image analysis in the CPU/GPU. While this can be a concern, a number of new developments are pushing for continuous vision applications on mobiles and very low power imaging sensors are emerging (see, e.g., [36]). Also, lower resolution tracking may still provide sufficient accuracy for our application, while reducing energy burden. Therefore, we expect that gaze tracking can be leveraged to improve uPLT in mobile devices.

Exploiting Saliency Models: Saliency models have been discussed in the previous section. A powerful approach could be to decrease reliance on actual gaze tracking, but rely instead on saliency models. In other words, inspecting Web pages via suitable modeling techniques could discover potential regions of user attention that could be a good proxy for gaze tracks. This approach is more scalable and would even apply to pages where gaze tracking data is not available. The challenge is that research on saliency models for Web pages is not yet mature. Our initial results show promise in leveraging gaze for improving uPLT; exploiting Web saliency models can significantly increase the deployability of our approach.

Systems Issues: There are a number of systems issues that need to be addressed to build a useful Web optimization based on gaze feedback. For example, a standardized gaze interface needs to be developed that integrates with the browser. The gaze support service (Figure 9) needs to adapt to changing nature of the Web contents and user interests. For example, a major event may suddenly change users' gaze behaviors on a news Web site even when the structure of the page remains the same.

Security and Privacy: There are additional security and privacy related concerns if gaze feedback is collected by Web sites or third party services. For example, it is certainly possible that gaze tracking could provide a significant piece of information needed to uniquely identify the user, even across devices. The use of eye tracking on the end-user's device exposes the user to hacks that could misuse the tracking data. Note that course-grained tracking information is sufficient for our task, but guaranteeing that only course-grain information is collected requires a hardened system design.

Gaze Tracking Methodology: Web page loads are a dynamic process. Therefore, collecting gaze data when the user looks only at the loaded Web page is not representative of the Web viewing experience. Instead, in this work, we collect gaze data *as* the page is being loaded. However, one problem is that, the gaze fixation is influenced by the Web object ordering. For instance, if objects that are important to the user are rendered later, a user may direct her gaze towards unimportant, but rendered objects. Our methodology partially alleviates the problem by capturing gaze only after the First Paint (§6) and even after OnLoad. As part of future work, we propose to track user gaze when the Web objects are loaded in different orders. By analyzing gaze under different object orderings, we hope to alleviate the problem of the Web page loading order influencing gaze tracks.

9 Conclusions

There has been a recent interest in making user experience the central issue in Web optimizations. Currently, user experience is divorced from Web page performance metrics. We systematically study the user-perceived page load time metric, or uPLT, to characterize user experience with respect to traditional metrics. We then make a case for using users' eye gaze as feedback to improve the uPLT. The core idea revolves around the hypothesis that Web pages exhibit high and low regions of collective interest, where a user may be interested in certain parts of the page and not interested in certain other parts. We design a system called WebGaze that exploits the regions of collective interest to improve uPLT. Our user study across 100 users and 45 Web pages shows that WebGaze improves uPLT compared to three alternate strategies for 73% of the Web pages.

Acknowledgements

We thank our shepherd Srinivasan Seshan and the anonymous reviewers for their invaluable feedback that greatly improved the presentation of this paper. We thank graduate students Sowmiya Narayan and Sruti Mandadapu for their help in bootsrapping the project and for their help with setting up the Web page videos. This work was supported by the National Science Foundation, through grants CNS-1551909 and CNS-1566260.

References

[1] 14 Rules for Faster-Loading Web Sites. http://stevesouders.com/hpws/rules.php.

[2] Above-the-fold time (AFT): Useful, but not yet a substitute for user-centric analysis. http://bit.ly/29MBmip.

[3] Alexa: a commercial web traffic data and analytics provider. http://www.alexa.com/.

[4] Document Object Model(DOM). https://www.w3.org/DOM/.

[5] Going Beyond OnLoad: Measuring Performance that matters. http://oreil.ly/2cpaUhV.

[6] Hick's law: On the rate of gain of information. https://en.wikipedia.org/wiki/Hick%27s_law.

[7] How to use eye tracking on samsung galaxy s7 and galaxy s7 edge. http://bit.ly/29LDiqj.

[8] Measuring the critical rendering path with Navigation Timing. https://developers.google.com/web/fundamentals/performance/critical-rendering-path/measure-crp?hl=en.

[9] Microworkers: Crowdworking/Crowdsourcing Platform. https://microworkers.com/.

[10] SPDY: An experimental protocol for a faster web. https://www.chromium.org/spdy/spdy-whitepaper.

[11] The onload property of the GlobalEventHandlers. https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onload.

[12] AKPINAR, M. E., AND YESILADA, Y. *Vision Based Page Segmentation Algorithm: Extended and Perceived Success*. Springer International Publishing, 2013, pp. 238–252.

[13] BELSHE M., PEON R., AND THOMPSON M. ED. Hypertext Transfer Protocol Version 2 (HTTP/2). https://tools.ietf.org/html/rfc7540.

[14] BERGAN T., PELCHAT S., B. M. Rules of thumb for HTTP/2 server push. http://bit.ly/2d4O1RN.

[15] BIXBY, J. Case study: The impact of HTML delay on mobile business metrics. http://bit.ly/2cbN22l, November 2011.

[16] BIXBY, J. 4 awesome slides showing how page speed correlates to business metrics at walmart.com. http://bit.ly/1jfACl2, February 2012.

[17] BOJKO, A. Using Eye Tracking to Compare Web Page Designs: A Case Study. In *Journal of Usability Studies* (2006), vol. 1, pp. 112–120.

[18] BROWN, M. A. Traffic control how to, overview of the capabilities and implementation of traffic control under linux. http://www.tldp.org/HOWTO/html_single/Traffic-Control-HOWTO/, 2006.

[19] BUSCHER, G., CUTRELL, E., AND MORRIS, M. R. What do you see when you're surfing?: Using eye tracking to predict salient regions of web pages. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, CHI '09, ACM, pp. 21–30.

[20] BUTKIEWICZ, M., MADHYASTHA, H. V., AND SEKAR, V. Understanding website complexity: Measurements, metrics, and implications. In *Proceedings on Internet Measurement Conference*, IMC '11, pp. 313–328.

[21] BUTKIEWICZ, M., WANG, D., WU, Z., MADHYASTHA, H. V., AND SEKAR, V. Klotski: Reprioritizing web content to improve user experience on mobile devices. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI '15, USENIX Association, pp. 439–453.

[22] CHROME DEBUG TEAM. Chrome remote debugging. https://developer.chrome.com/devtools/docs/debugger-protocol.

[23] CIOTTI, G. 7 marketing lessons from eye-tracking studies. https://blog.kissmetrics.com/eye-tracking-studies/.

[24] ERASLAN, S., YESILADA, Y., AND HARPER, S. Eye tracking scanpath analysis techniques on web pages: A survey, evaluation and comparison. *Journal of Eye Movement Research vol. 9*, No. 1 (2015), 1–19.

[25] FERHAT, O., AND VILARIÑO, F. Low cost eye tracking: The current panorama. *Computational Intelligence and Neuroscience vol. 3*, No. 2 (2016), 1–14.

[26] GAO, R., UCHIDA, S., SHAHAB, A., SHAFAIT, F., AND FRINKEN, V. Visual saliency models for text detection in real world. *PloS ONE vol. 9*, No. 12 (2014), 1–20.

[27] GAZEPOINTER. Control mouse cursor position with your eyes via webcam. http://gazepointer.sourceforge.net/.

[28] GOFERMAN, S., ZELNIK-MANOR, L., AND TAL, A. Context-aware saliency detection. In *Transactions on Pattern Analysis and Machine Intelligence*, vol. 34 of *PAMI '12*, IEEE, pp. 1915–1926.

[29] GOOGLE DEVELOPERS. PageSpeed Tools: The PageSpeed tools analyze and optimize your site. https://developers.google.com/speed/pagespeed/.

[30] GOVINDAN, R. Modeling HTTP/2 speed from HTTP/1 traces. In *Proceedings of 17th International Conference on Passive and Active Measurement*, vol. 9631 of *PAM '16*, Springer, p. 233.

[31] GRIGORIK, I. Analyzing critical rendering path performance. http://bit.ly/1ORhrNj.

[32] IMMS, D. Speed index: Measuring page load time a different way. http://bit.ly/2dbuUpr, September 2014.

[33] JOHN FINDLAY AND ROBIN WALKER. Human saccadic eye movements. http://www.scholarpedia.org/article/Human_saccadic_eye_movements.

[34] KIT EATON. How one second could cost amazon 1.6 billion in sales. http://bit.ly/1Beu9Ah.

[35] KRAFKA, K., KHOSLA, A., KELLNHOFER, P., KANNAN, H., BHANDARKAR, S., MATUSIK, W., AND TORRALBA, A. Eye tracking for everyone. In *Conference on Computer Vision and Pattern Recognition*, CVPR '16, IEEE.

[36] LIKAMWA, R., PRIYANTHA, B., PHILIPOSE, M., ZHONG, L., AND BAHL, P. Energy characterization and optimization of image sensing toward continuous mobile vision. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, ACM, pp. 69–82.

[37] LU, F., SUGANO, Y., OKABE, T., AND SATO, Y. Inferring human gaze from appearance via adaptive linear regression. In *Proceedings of the International Conference on Computer Vision*, ICCV '11, IEEE, pp. 153 –160.

[38] MOZILLA DEVELOPER NETWORK. Remote debugging. https://developer.mozilla.org/en-US/docs/Tools/Remote_Debugging.

[39] NETRAVALI, R., GOYAL, A., MICKENS, J., AND BALAKRISHNAN, H. Polaris: Faster page loads using fine-grained dependency tracking. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation*, NSDI '16, USENIX Association, pp. 123–136.

[40] O'CONNELL, C. Eyetracking and web site design. https://www.usability.gov/get-involved/blog/2010/03/eyetracking.html, March 2010.

[41] PAPOUTSAKI, A., SANGKLOY, P., LASKEY, J., DASKALOVA, N., HUANG, J., AND HAYS, J. Webgazer: Scalable webcam eye tracking using user interactions. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, IJCAI '16, AAAI, pp. 3839–3845.

[42] PETROVICH, M. Going beyond onload: Measuring performance that matters. http://oreil.ly/2cpaUhV, October 2013.

[43] RYOO, J., YUN, K., SAMARAS, D., DAS, S. R., AND ZELINSKY, G. J. Design and evaluation of a foveated video streaming service for commodity client devices. In *Proceedings of the 7th International Conference on Multimedia Systems*, MMSys '16, ACM, pp. 6:1–6:11.

[44] SHARMA, G., JURIE, F., AND SCHMID, C. Discriminative spatial saliency for image classification. In *Conference on Computer Vision and Pattern Recognition*, CVPR '12, IEEE, pp. 3506–3513.

[45] SHEN, C., HUANG, X., AND ZHAO, Q. Predicting eye fixations on webpage with an ensemble of early features and high-level representations from deep network. *IEEE Transactions on Multimedia vol. 17*, No. 11 (2015), 2084–2093.

[46] SHEN, C., AND ZHAO, Q. *Webpage Saliency*. Springer International Publishing, 2014, pp. 33–46.

[47] SIVAKUMAR, A., PUZHAVAKATH NARAYANAN, S., GOPALAKRISHNAN, V., LEE, S., RAO, S., AND SEN, S. Parcel: Proxy assisted browsing in cellular networks for energy and latency reduction. In *Proceedings of the 10th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '14, ACM, pp. 325–336.

[48] SMI Eye Tracking Glasses 2 Wireless. http://bit.ly/29YWaDa.

[49] SOUDERS, S. Moving beyond window.onload(). http://bit.ly/1VzsdME, May 20.

[50] STILL, J. D., AND MASCIOCCHI, C. M. A saliency model predicts fixations in web interfaces. In *5th International Workshop on Model Driven Development of Advanced User Interfaces*, MDDAUI '10, pp. 25–28.

[51] SUGANO, Y., MATSUSHITA, Y., AND SATO, Y. Learning-by-synthesis for appearance-based 3d gaze estimation. In *Conference on Computer Vision and Pattern Recognition*, CVPR '14, IEEE, pp. 1821–1828.

[52] THE CHROMIUM PROJECTS. QUIC, a multiplexed stream transport over UDP. `http://bit.ly/2cDBKig`.

[53] VARELA, M., SKORIN-KAPOV, L., MÄKI, T., AND HOSSFELD, T. QoE in the Web: A dance of design and performance. In *7th International Workshop on Quality of Multimedia Experience*, QoMEX '15, pp. 1–7.

[54] VARVELLO, M., BLACKBURN, J., NAYLOR, D., AND PAPAGIANNAKI, K. Eyeorg: A platform for crowdsourcing web quality of experience measurements. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, ACM, pp. 399–412.

[55] VICK, R. M., AND IKEHARA, C. S. Methodological issues of real time data acquisition from multiple sources of physiological data. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, HICSS '03, pp. 1–7.

[56] VIOLA, P., AND JONES, M. J. Robust real-time face detection. *International Journal of Computer Vision vol. 57*, No. 2 (2004), 137–154.

[57] Peripheral vision. `https://en.wikipedia.org/wiki/Peripheral_vision`.

[58] WANG, H.-C., AND POMPLUN, M. The attraction of visual attention to texts in real-world scenes. *Journal of Vision vol. 12*, issue 6 (2012), 26–26.

[59] WANG, P., WANG, J., ZENG, G., FENG, J., ZHA, H., AND LI, S. Salient object detection for searched web images via global saliency. In *Conference on Computer Vision and Pattern Recognition*, CVPR '12, IEEE, pp. 3194–3201.

[60] WANG, X. S., BALASUBRAMANIAN, A., KRISHNAMURTHY, A., AND WETHERALL, D. Demystifying page load performance with WProf. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, USENIX Association, pp. 473–486.

[61] WANG, X. S., BALASUBRAMANIAN, A., KRISHNAMURTHY, A., AND WETHERALL, D. How speedy is SPDY? In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI '14, USENIX Association, pp. 387–399.

[62] WANG, X. S., KRISHNAMURTHY, A., AND WETHERALL, D. Speeding up web page loads with shandian. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation*, NSDI '16, USENIX Association, pp. 109–122.

[63] WebPagetest: website performance testing service. `http://www.webpagetest.org/`.

[64] YANG, J., AND YANG, M.-H. Top-down visual saliency via joint CRF and dictionary learning. In *Conference on Computer Vision and Pattern Recognition*, CVPR '12, IEEE, pp. 2296–2303.

RAIL: A Case for Redundant Arrays of Inexpensive Links in Data Center Networks

Danyang Zhuo[†] Monia Ghobadi Ratul Mahajan Amar Phanishayee
Xuan Kelvin Zou Hang Guan[*] Arvind Krishnamurthy[†] Thomas Anderson[†]

Microsoft Research [†] University of Washington [*] Columbia University

Abstract— While there are many proposals to reduce the cost of data center networks (DCN), little attention has been paid to the role played by the physical links that carry packets. By studying over 300K optical links across many production DCNs, we show that these links are operating quite conservatively relative to the requirements in the IEEE standards. Motivated by this observation, to reduce DCN costs, we propose using transceivers—a key contributor to DCN cost—beyond their currently specified limit. Our experiments with multiple commodity transceivers show that their reach can be "stretched" 1.6 to 4 times their specification. However, with stretching, the performance of 1–5% of the DCN paths can fall below the IEEE standard. We develop RAIL, a system to ensure that in such a network, applications only use paths that meet their performance needs. Our proposal can reduce the network cost by up to 10% for 10Gbps networks and 44% for 40Gbps networks, without affecting the applications' performance.

1 Introduction

The relentless demand for capacity in data center networks (DCNs) makes their cost a primary metric of interest. Network cost optimization must consider all main contributors, including switches, links, and power. Several innovations in DCN architecture [18, 35, 55, 21, 24, 45, 37] have transformed the industry and reduced the cost of building and operating DCNs.

Yet the physical layer of DCNs (i.e., links) has received little attention, and it is being engineered using conservative practices that lead to a high cost. In modern DCNs, all inter-switch links tend to be optical, and the design objective is that *every* link should have a bit error rate (BER) lower than 10^{-12}.[1] To meet this objective, when manufacturers rate transceivers—devices that convert signals between electrical and optical domains—for a certain link length (e.g., 300m), they assume worst case operating conditions (e.g., temperature, signal attenuation due to connectors). These worst-case conditions are rare, and consequently, the vast majority of the links are significantly over-engineered—the optical signal quality is much higher than what is needed to support 10^{-12} BER.

To confirm and quantify physical layer over-engineering in today's DCNs, we conduct what to our knowledge is the first large-scale study of operational optical links. We analyze over 300K links across more than 20 data centers of a large cloud provider over a period of 10 months. We find a remarkably conservative state of affairs—99.9% of the links have incoming optical signal quality that is higher than the minimum threshold for 10^{-12} BER, while the median is 6 times higher!

This over-engineering is expensive—transceivers account for 48-72% of total DCN cost depending on link speed and link length distribution (§7)—and reducing it can lower network costs. While there are multiple ways to do so, we explore an approach that does not require any changes to existing hardware. Inspired by the approach of running data centers at higher temperatures than manufacturers' specifications for hardware [29], we suggest DCN operators use transceivers for link lengths that are greater than manufacturers' specifications. This approach can lower costs because transceivers with shorter length specification tend to be cheaper; thus, cheaper transceivers can be used for many links that ostensibly need more expensive ones today.

However, because of transceiver "stretching," some links in the DCN may have BER higher than 10^{-12}. Traffic on such *gray* links will experience higher loss rates because more packets will have bit errors (which switches will drop). But the application and path diversity of modern DCNs suggest the overall impact on application performance is likely to be negligible. Many applications do not need BER of 10^{-12} and can be carried over gray links without hurting their performance. Moreover, since the network has many paths between pairs of top-of-rack switches, applications requiring high reliability can be well supported by being routed through low-loss paths.

Two questions remain about our approach: *i*) how much can cost be reduced by "stretching" transceivers beyond their specifications? And *ii*) how can we ensure applications only use paths that meet their loss tolerance? To answer the first question, we perform extensive simulations and experiments with transceivers under realistic conditions. We show that depending on the technology (10 or 40 Gbps) and the desired upper bound on the fraction of gray

[1] The BER requirement of 10^{-12} was standardized by the IEEE in 2000 [11] and is likely a holdover from the telecom world. In reality, few DCN applications today need that level of BER. RDMA is perhaps the most BER-sensitive application today, and a BER of 10^{-10} suffices for it [60]. However, for the purposes of this paper, we assume some (future) applications will need 10^{-12} BER.

paths (1% or 5%), current transceivers can be stretched from 1.6 to 4 times their specified length. At these design points, for a fat tree topology, and depending on the link length distribution, the cost of the network can be lowered by up to 10% for 10Gbps and 44% for 40Gbps networks.

We answer the second question by designing RAIL. It is a practical system that builds multiple virtual topologies on the same physical topology, where the class of the topology offers a bound on the maximum packet error rate (i.e., grayness) on any path in it. The first-class topology does not have any gray paths; hence, it offers the same path packet error rate guarantee as current DCN designs. Other classes increasingly use more gray links. Each application uses the virtual topology that meets its needs. Thus, loss-tolerant applications use virtual topologies that may have more gray paths. To support applications, such as large transfers that are otherwise loss-tolerant but suffer when the transport protocol (e.g., TCP) is sensitive to losses, RAIL uses a transparent coding-based error correction scheme. We develop an efficient algorithm to compute virtual topologies that leverages the hierarchical topological structure of DCNs (e.g., Clos). RAIL is easily deployable as it requires no changes to the switch or transceiver hardware.

We evaluate RAIL using simulations-based analysis and a testbed. Even at the maximum stretched reach level we consider, we find 95% of all paths are as reliable as today. Furthermore, RAIL protects loss-sensitive applications from gray paths.

2 Optical Links' Performance in the Wild

In today's DCNs, switches are arranged in a multi-tier topology such as a fat tree or folded Clos [55, 18]. While the switches are electrical, the links between them are optical because optical links are more cost-efficient at higher bandwidths and distances.

An optical link has transceivers at each end connected via a fiber cable. Transceivers plug into switches to convert electrical signals nto optical signals and vice versa. Figure 1 shows a transceiver and its internal components. It has a transmit and a receive pipeline, each attached to an independent fiber. The transmit side has a laser that emits light and a modulator that modulates it according to the electrical input from the switch. The receive pipeline decodes incoming optical signals using a photodetector and an amplifier.

The performance of an optical link is quantified using BER, the probability of a single bit being corrupted at the receiver. Corruption happens when the optical power gap between bits 0 and 1 is too small to be reliably differentiated at the receiver. The gap can be low because of poor signal characteristics at the transmitter (e.g., a poor-quality laser that spreads light over a wide spectrum) or because of attenuation and dispersion as light travels through fiber. Attenuation refers to a reduction in average signal power, e.g., caused by the optical connectors

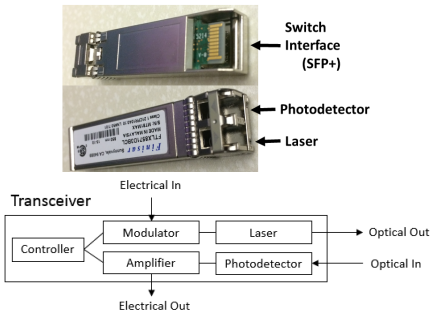


Figure 1: *The schematic of an optical transceiver.*

attaching fibers to transceivers. Dispersion is the distortion of the shape of the signal.

To simplify manufacturing and deployment, IEEE standardizes transceiver characteristics. Apart from wavelength (e.g., 850, 1310, 1510 nm) and transmission speed (e.g., 10, 40, 100 Gbps), an important characteristic of each standard is the maximum reach at which a transceiver is guaranteed to deliver a BER of 10^{-12} or better.

Two main categories of optical technologies are used in DCNs today: multi-mode and single-mode. Multi-mode transceivers are cheaper because they have relaxed constraints on laser spectral width and coupling of laser with fiber. But they also have a shorter reach because they suffer from modal dispersion, i.e., signal distortion due to differing path lengths taken by light waves in the fiber.

2.1 Data Set

To shed light on the performance of the optical layer in today's DCNs, we build a monitoring system that uses SNMP optical MIB [12] to poll optical performance metrics from transceivers every five minutes. Our system runs in multiple production data centers and collects transceivers' transmit power (TxPower), receive power (RxPower), temperature, transmit bias current, and supply voltage. The TxPower and RxPower values reported by the transceivers are average (across bits 0 and 1) signal power values.

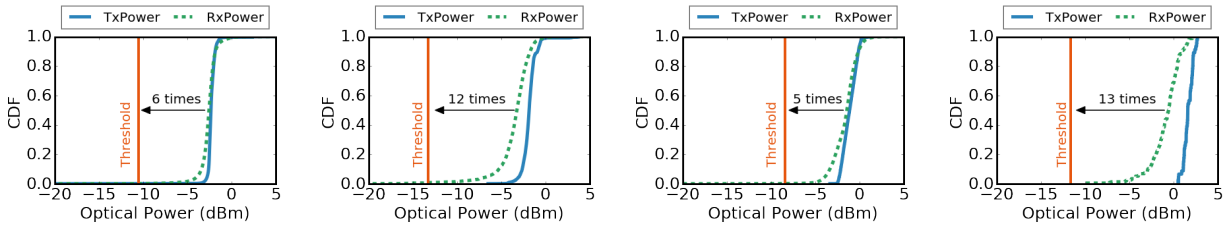
The data we report in this paper were collected over ten months from Nov 2015 to Aug 2016. They cover over 300K links (600K transceivers), with a mix of multi- and single- mode transceivers with 10, 40 and 100Gbps speeds.

2.2 Optical Power Levels Are Too Good

We begin our analysis by studying optical transmit and receive power levels. RxPower is a key indicator of optical layer performance. Modulo significant dispersion, it directly determines BER.[2] To keep BER under 10^{-12}, RxPower should be above the receiver sensitivity required by the IEEE standard.

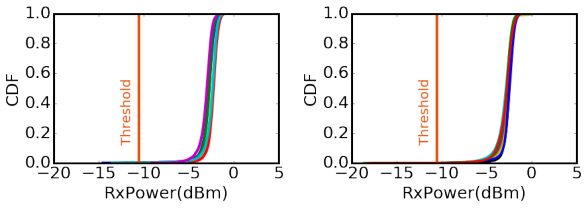
Degree of power over-engineering. Figure 2 shows the CDF of average TxPower and RxPower of all transceivers

[2]Most transceivers today do not report BER. Switches report packet error rate, but this measures the combined impact of errors from all sources, including electrical components.



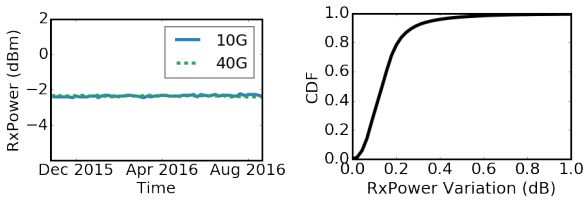
(a) 10Gbps multi-mode (b) 10Gbps single-mode (c) 40Gbps multi-mode (d) 100Gbps single-mode

Figure 2: *CDF of TxPower and RxPower for different speeds and modes. The vertical lines are the RxPower threshold needed to keep BER under* 10^{-12}. *Almost all links across all technologies have RxPower above the threshold.*



(a) *Different manufacturers* (b) *Different data centers*

Figure 3: *CDF of RxPower of 10Gbps multi-mode transceivers across five transceiver manufacturers and ten different data centers. Over-engineering is not limited to one manufacturer or data center.*



(a) *RxPower vs. time.* (b) *(max - min) RxPower*

Figure 4: *RxPower of individual transceivers has small variations over time. 10Gbps and 40Gbps lines are overlapping in the left graph.*

separated by their speed and mode (10Gbps multi- and single- mode, 40Gbps multi-mode and 100Gbps single-mode). The vertical line in each plot is the receive power threshold to keep BER under 10^{-12}. We see that across the board nearly all links (99.9%) have RxPower above the threshold. In fact, 99% of transceivers are at least 5 dB (i.e., 3 times) above it. Thus, when we use the IEEE standard as our basis of comparison, we see the vast majority of links are greatly over-engineered from an optical power perspective.

Over-engineering across transceiver models. We find this degree of over-engineering is consistent across transceiver manufacturers. Our data include over 50 transceiver models across various vendors. Figure 3a plots the CDF of RxPower of 10Gbps multi-mode transceivers across five transceiver manufacturers with more than 1000 transceivers inside one data center. For confidentiality, we do not report the name of our manufacturers. All five manufacturers exhibit similar RxPower distributions, and over-engineering is not tied to one manufacturer.

Over-engineering across data centers. We also study whether over-engineering varies by data center. Figure 3b plots power levels of the ten largest data centers in our dataset. Each color represents a data center. We see that RxPower distributions and thus over-engineering levels are similar for every data center.

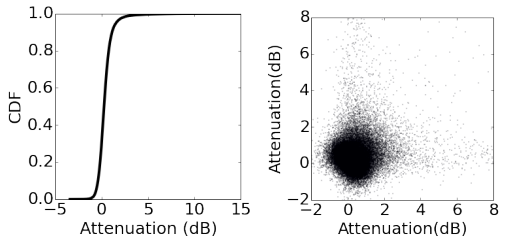
Optical power variation over time. Over the course of our study, we found power levels of a link vary little over time. Figure 4a illustrates RxPower over time for two sample transceivers. As shown, for each transceiver, RxPower remains mostly stationary over time. Figure 4b plots the CDF of maximum minus minimum RxPower value for all transceivers. The figure shows 78% of the links have a variation below 0.2 dB, and over 99% have a variation below 0.8 dB. Thus, links with high RxPower have consistently high RxPower, instead of power levels dropping intermittently (which would cause high BER during those times if we were to reduce their over-engineering).

2.3 Understanding Low Optical Power

In Figure 2, unlike TxPower, RxPower has a long tail, suggesting that a small fraction of links experience high attenuation and thus low RxPower. Figure 5a shows this effect directly by plotting the CDF of attenuation for all multi-mode links; single-mode results are similar. We compute attenuation as TxPower at the sender minus RxPower at the receiver. The figure shows most links have attenuation close to zero—the median is 0.26dB—but 0.44% have attenuation higher than 5dB.

Initially, we thought high attenuation would correspond to links that are long or have many optical connectors. To confirm this supposition, we studied *attenuation symmetry* between the two directions of a bi-directional optical link. If link length or connector count are to blame, because these factors are identical in both directions, high attenuation should be symmetric. Figure 5b shows a scatter plot where the two axes represent attenuation levels in different directions. We see that attenuation is low and symmetric for most links.[3] But it is asymmetric when it is high. When one side has high attenuation, the other side does not.

[3] Attenuation levels below 0 are due to (unbiased) calibration error



(a) *CDF of Attenuation* (b) *Bi-directional Attenuation*

Figure 5: *Attenuation is nearly zero and symmetric for most links. Interestingly, high attenuation is asymmetric.*

This behavior suggests that high attenuation does not stem from link length or connector count but due to poor or dirty connectors or fiber damage—connectors and fibers are different in the two directions. To confirm this hypothesis, we analyzed hundreds of repair records for poor optical links. We found two main root causes for low RxPower: (i) dirty fiber connectors (which must be cleaned to fix the problem); (ii) damage to the fiber's cladding, the outside layer of the fiber that protects the actual core. The finding that high attenuation rarely stems from long links is consistent with our later experiments showing current transceivers have low BER even on links that are longer than their specification.

Dependence on link location. We also find that low RxPower links are scattered across the data center uniformly and randomly, and they are not correlated with a specific switch brand or topology tier. To confirm, we compute the percentage of switches having links with RxPower at the bottom 0.01% of Figure 2. We then uniformly and randomly select 0.01% of links with any RxPowers and again compute the percentage of switches to which they belong. If the two numbers match, it suggests links with RxPower at the tail are scattered uniformly and randomly across switches. We repeat this analysis for 100 values between 0.01%-tile and 1%-tile tail and observe the same result. Figure 6 shows that the numbers based on this independence assumption closely match the data. If some switches were more likely to have links with low power levels, the "Low RxPower data" curve would be consistently lower than the "Uniform Random" curve. This observation, together with our observation on the root causes of high attenuation, suggests that dirty connectors and damaged fiber can show up anywhere in the data center.

3 Reducing Over-engineering and Cost

Our measurements above reveal that the optical layer of DCNs today is heavily over-engineered. This over-engineering does not come for free; DCNs are expensive to build, with transceivers accounting for 48-72% of the network cost, depending on link speed and link length

in TxPower and RxPower sensors. The reported power is within $\pm$ 2 dB of the actual value.

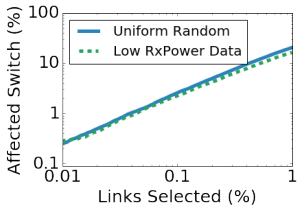


Figure 6: *Links with low RxPower are uniformly and randomly scattered across switches.*

distribution (Table 1). We shared our measurement results with experts in transceiver manufacturing companies and learned of many possibilities to lower the price of transceivers (e.g., changing transceiver hardware, relaxing test requirements, reducing labor and packaging cost).

However, we also learned that immediate realization is difficult because of two inter-twined challenges. First, transceiver manufacturers cannot reliably estimate cost savings without carefully crafting and optimizing the entire manufacturing chain, and they are reluctant to do so without standardization or firm commitments from DCN operators. Second, since any method of reducing over-engineering can cause some gray links (with BER higher than 10^{-12}), DCN operators are reluctant to commit unless the cost savings are known in advance to be high, and there is a way to protect sensitive applications against gray links.

In this paper, we solve both challenges. First, we demonstrate immediate cost savings are possible by allowing commodity transceivers to be "stretched," that is, using them for longer than their currently-specified distances. Second, we devise a system for routing and forwarding packets where some links may be gray. Leveraging application and path diversity in DCNs enables applications to use paths according to their loss-tolerance (§4).

Below, we explain how stretching reduces cost and show how to engineer a stretched network.

Cost reduction through stretching. Figure 7 shows the price and specified reach of different standard transceiver technologies available in the market for 10, 40, and 100Gbps. The price of 10 and 40Gbps transceivers represents the average across three volume retailers [14, 13, 7] and the price of 100Gbps transceivers is from one retailer [10]. As shown, the standard includes discrete reaches, with the price of transceivers increasing as reach is increased. Longer reach requires more expensive components (e.g., narrow spectrum laser, cooling module) and manufacturing processes.

Figure 8 illustrates how stretching transceivers reduces cost. The solid line is price for standard reach, and the dashed line shows the stretched reach. When the reach is stretched from R_1 to R'_1, we don't have to pay more to use an R_2-rated transceiver for distances between R_1 and R'_1 and, hence, we can save cost. The exact savings depends on how much transceivers can be stretched and the relative

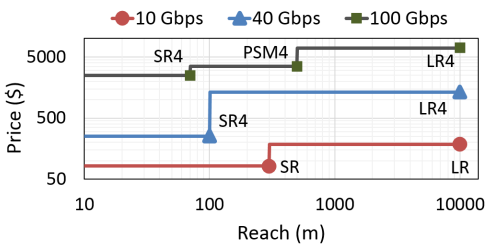


Figure 7: *Price and reach of standard transceivers for 10Gbps, 40Gbps and 100Gbps technologies. Both X and Y axes are in log scale. The label beside each data point is the name of the technology. PSM4 [16] is not yet an IEEE standard but is an agreement between 12 companies.*

pricing of different options. We quantify the savings for available standard technologies in §7.

We don't expect transceivers' lifetimes to be shortened by stretching because lasers are designed for a long life-time, and TxPower does not decrease over time (96.7% of transceivers' TxPower does not change more than 0.2 dB over our 10-month measurement period).

Determining how much to stretch. To determine how much transceivers in a DCN can be safely stretched, we use a metric called *network reliability bound* (NRB). NRB is a lower bound on the expected fraction of paths that the DCN administrator desires to be good (i.e., those where all links have BER below 10^{-12}). We compute the maximum stretch for a transceiver based on NRB and *i*) the maximum number of links in a DCN path, and *ii*) the BER distribution expected for different stretch levels. We show in §7 how to compute this distribution using expected attenuation distribution.

This computation is best illustrated in an example. Suppose our goal is for NRB to be 95%, which can be translated to each path being good with a probability of at least 95%. In a 3-stage Clos network, where the maximum number of hops is four, this goal can be met if each link is good with probability of $\sqrt[4]{95\%} = 99.7\%$ (assuming that links being good or bad is independent of location, as we showed in §2.3). From the BER distribution for different stretch levels, we can now determine the stretch at which 99.7% of the links will be good. We use this value as the maximum stretch for a transceiver. In reality, the network will have more than 99.7% good links because many links where "stretched" transceivers will be used are shorter than the maximum stretch.

Need for software layer protection. NRB enforces a lower bound on the fraction of good paths, allowing a small fraction of gray paths. To preserve application performance, a naive solution is to simply turn gray links off and thus remove all gray paths. But even if the fraction of gray links is small, turning them off can result in 20 to 50% capacity loss for certain ToR-pairs (§7). Instead, we propose that a software system be used

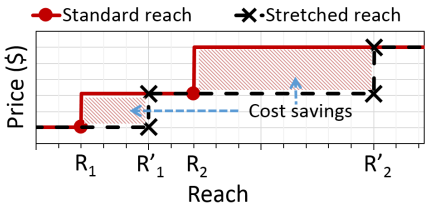


Figure 8: *Illustration of how stretching transceivers reduces cost. When R1-rated transceiver is stretched from R_1 to R'_1, we do not have to use (the more expensive) R_2-rated transceiver for link lengths between R_1 and R'_1.*

to effectively utilize such gray links, while protecting loss-sensitive applications. The design of RAIL, described next, accomplishes this goal.

4 Overview of RAIL

RAIL is a system for routing and forwarding in a DCN, where *i*) links have a range of link packet error rate (LPER); and *ii*) applications have different requirements for path packet error rate (PPER). In such a network, RAIL ensures an application is routed only through *paths* with its desired reliability level (or better). Thus, sensitive applications (i.e., RDMA-based ones) are routed only along the most reliable paths while tolerant ones (i.e., UDP-based ones) can be routed through any path. RAIL is a general solution that is agnostic to why gray links occur; they could be cause by any method of reducing over-engineering, including stretched transceivers, cheaper hardware or cheaper fiber.

We want our solution to be: *i*) readily deployable, i.e., requiring minimal changes to current infrastructure; and *ii*) practical, i.e., having low overhead and complexity. To appreciate these constraints, we consider two extreme design possibilities with respect to how much hosts need to know about the network. The first is source routing, in which hosts are responsible for selecting paths (i.e., sequence of links to the destination) through the network that meet application needs. The challenge with this approach is that hosts will need an up-to-date view of network topology and LPER. With hundreds of thousands of links and frequent link failures [34], maintaining such a view at every host is highly complex.

In the second design, hosts are not told anything about the network (as is the case today). To deliver reliable communication, they use network coding transparently; i.e., application traffic to a given destination is coded such that it can withstand some fraction of loss. The problem with this approach is that it sets up an unwanted trade-off between coding overhead and the loss rate experienced by traffic. Hosts do not know in advance which path a given flow might take (due to ECMP routing). If they encode every flow to a level that guarantees the most sensitive applications do not suffer (i.e., encoding based on the least reliable path), the coding overhead will be onerous for most paths. If they encode based on the average path

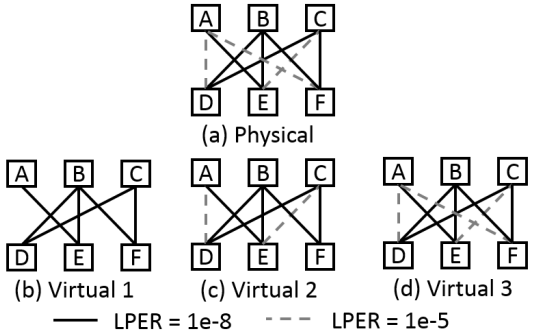


Figure 9: *Example of RAIL's virtual topologies. Solid lines are links having* 10^{-8} *LPER. Dashed gray lines are links having* 10^{-5} *LPER. RAIL maintains three virtual topologies (b, c, d) depending on Path Packet Error Rate (PPER).*

loss rate, sensitive applications that traverse worse paths will suffer. It is possible to adapt by learning path loss rate based on what is actually experienced by a flow, but this is complicated by the fact that most flows are short and do not generate enough packets for robust learning.

To be deployable and practical, RAIL explores a design point in the middle. On the same physical topology, it builds *k virtual topologies*. Each topology falls in a different *class* representing a bound on the worst Path Packet Error Rate (PPER) in the topology. PPER represents the packet corruption probability for a complete ToR-to-ToR path and is derived from each hop's LPER. The first-class topology is made exclusively of the most reliable paths in the physical topology, the second-class topology permits all paths with PPER less than a bound, and the *k*-th class topology may include all paths in the physical topology. Figure 9 illustrates this concept with *k*=3, that is, three virtual topologies. Within a virtual topology, routing and forwarding will follow the same protocol as DC operators prefer today for their physical topology (e.g., ECMP over equal hop paths). Figure 9a shows the physical topology with two types of links: some with LPER 10^{-8} and three gray links with LPER 10^{-5}. RAIL creates three virtual topologies shown in Figure 9b, c, and d. Virtual topology (b) guarantees PPER of 2×10^{-8}. Virtual topology (c) guarantees PPER of 1×10^{-5} since its worst paths are A-D and C-E. Virtual topology (d) guarantees PPER of 2×10^{-5} since its worst path is D-A-F. Thresholds of these virtual topologies are configurable.

Virtual topologies are exposed to end hosts as different (virtual) interfaces. Applications (or the hypervisor on their behalf) bind to the interface that reflects their reliability criteria. Thus, the most-sensitive applications (e.g., RDMA-based ones) may bind to the interface for the first-class topology, TCP flows may bind to the second-class topology, and bulk transfers may bind to the *k*-class topology. Beyond making this choice, hosts are not aware of RAIL.

Applications that bind to less reliable topologies may be already robust to small amounts of loss they may experience, or they can be made that way by using a version of TCP that is robust to corruption-based losses [22, 44]. If this is not the case, however, RAIL includes a module that uses coding to transparently enhance traffic reliability.[4] Designing this module is an easier task than the coding-based option outlined above; flows going over less reliable topologies are likely longer, giving our module a chance to learn the path being used and select an appropriate coding level.

In the following section, we describe RAIL's design in more detail. As will become clear, it requires no changes to existing switches and only small changes to the host software, and then only if the error correction module is used.

5 RAIL in Detail

This section describes virtual topologies, routing and forwarding, and error correction mechanisms in RAIL. It also provides guidelines on configuring RAIL.

5.1 Virtual Topology Construction

Virtual topologies in RAIL share the same physical fabric but offer different guarantees for the maximum packet corruption rate along any of their paths. Our task is to build K virtual topologies, where the k-th topology offers the worst-case PPER of L_k. K and $\{L_k\}$ are selected by DCN operators based on their applications (see §5.4).

These topologies are built by the RAIL controller, which maintains an up-to-date view of each link's LPER by polling the switches. Since link qualities are relatively static (§2), maintaining such a view is straightforward. In addition to providing a worst-case PPER guarantee, we want each virtual topology to include as many links as possible to maximize its capacity. Because optimally finding such a topology is computationally expensive, we resort to a fast, greedy algorithm. Speed is important because, while link qualities are relatively static, links do fail frequently [34], and we need to recompute virtual topologies when links fail or recover.

Our algorithm has K rounds, one per topology. In each round, it starts with a set of candidate links for the topology and iteratively removes low reliability links until the required guarantee can be met. We start with the K-th class (least reliable) topology. For it, all links (that are currently alive) are initial candidates. We find the ToR-to-ToR path with the worst PPER, and if that PPER is higher than the required bound, we remove the link (on the path) with the worst LPER. Such link-removal iterations are repeated until the worst-case path meets the required bound. We then begin the next round, for the topology that is one class lower, starting with links not removed in the previous round.

[4] Retransmitting lost packets is another (more efficient in terms of byte overhead) way to improve traffic reliability. We explore coding because it offers a faster, proactive way to recover lost data.

The speed of the above algorithm depends on how quickly we can check whether a topology meets a PPER requirement. Tracking every path's PER takes at least $O(n^3)$ in a n-ToR Clos topology (as there are $O(n^2)$ ToR pairs with $O(n)$ paths between each pair). Such running time is computationally infeasible on realistic network controllers. Our algorithm speeds up this running time to $O(n \log n)$ and is linear in the number of links in the topology. The intuition is that because of a DCN's highly structured topology and its simple routing scheme, we can track the worst path without tracking the PPER on every path.

Our algorithm goes through each switch exactly once and computes three values on each switch, from the bottom stage all the way to the top stage. The three values computed on switch s are (1) up_s: PPER for the worst monotonic upward path from any ToR to s, (2) $down_s$: PPER for the worst monotonic downward path from s to any ToR, and (3) top_s: PPER for the worst up-down path for which s is the highest stage switch.

Definition 1. *Children of switch s are the set of switches on a lower stage than s, with direct links to s.*

$up_s \leftarrow 0$
$down_s \leftarrow 0$
$top_s \leftarrow 0$
for $c \in Children(s)$ **do**
  $up_s \leftarrow max(up_s, 1-(1-up_c)(1-LPER_{c\to s})$
  $down_s \leftarrow max(down_s, 1-(1-LPER_{s\to c})(1-down_c))$
end
for $c,d \in Children(s),\ c \neq d$ **do**
  $tops_s \leftarrow max(top_s, 1-$
    $(1-up_c)(1-LPER_{c\to s})(1-LPER_{s\to d})(1-down_d))$
end

Algorithm 1: Update up_s, $down_s$, top_s on switch s.

After those three numbers are calculated, our algorithm outputs the worst path by picking the worst top_s among all switches in the network. The worst PPER is simply $max_{s \in all\ switches}(top_s)$. The correctness proof and running-time analysis appear in Appendix B.

5.2 Routing and Forwarding

To simplify routing and forwarding, we use a non-overlapping IP address space within each virtual topology. We configure switches such that, when routing or forwarding for a topology, they ignore links that are not part of that topology. The exact mechanism will depend on the routing paradigm used by the data center. If the data center uses a distributed protocol such as BGP to compute paths [41], we configure BGP to not announce prefixes for a virtual topology over links that are not part of it. No RAIL-specific changes are made to switch software, and they will forward packets as they do today (e.g., ECMP).

If the data center uses an SDN controller to centrally compute forwarding paths, we can either instantiate one controller per virtual topology or use one network-wide controller programmed to not use certain links for given prefixes.

Our approach may create uneven load on links because different links are part of different topologies. Load is uneven even without our modifications, however, and traffic engineering is required to balance it [59, 19, 52]. We leave the task of extending traffic engineering to account for virtual topologies for future work.

5.3 Error Correction

When the application or transport protocol is not robust to small amounts of PPER (corruption-based loss), RAIL's error correction module can be used to guarantee high performance in exchange for slight bandwidth overhead. This module is completely transparent to applications.

As argued earlier, given the diversity of PPERs across paths, it is important that error correction be based on the PPER of each path, rather than being guided by the worst-case or average PPER in the virtual topology. Recall that because of ECMP-hashing, hosts are not aware of the path taken by a flow.

RAIL's error correction module learns the PPER in two steps. First, as soon as a new flow starts, the source host sends a `traceroute` probe with a header (5-tuple) identical to that of the flow. This probe reveals the path taken by the flow. We use special DSCP bits in the IP header of the probe packet to indicate to the destination host module that it should not deliver the packet to the application. Second, the error correction module queries RAIL's controller for PPER of the path.

Ideally, the error correction module should be able to use bit-level forward error correction (FEC) for each packet. However, this approach does not work in practice because today's switches drop packets when CRC checksum fails. As we are seeking an immediately deployable solution, we do not consider this option. The main benefit is that bit-level FEC has low coding overheads. As we show in §7, the bandwidth overhead of our error correct module is already low enough.

RAIL sends "parity" packets after every n data packets, where n is based on the path packet loss rate (see below). We use XOR encoding because it is lightweight and known to be effective [53]. That is, after every n data packets, the sender sends a packet whose content is the XOR of the previous n packets. In this coding scheme, as long as n out of the $n+1$ packets are successfully delivered, the receiver can recover the original n packets. Losing two or more packets within a group of $n+1$ packets results in data loss.

If PPER is p, the probability of having two or more losses among $n+1$ packets is $1-(1-p)^{n+1}-(n+1)p(1-p)^n$. We pick n such that this probability is lower than the desired post-recovery loss probability t (experienced by applications). Any path with $p < t$ does not use any error correction. To show an example of computing n, we first quantify t for a particular transport. Suppose TCP's performance degrades when

the loss rate is above 0.1%; therefore, we can pick t=0.1%. For a path loss rate of $p = 0.3\%$, we would choose n to be 14 so that the post-recovery loss rate is again $0.092\% < 0.1\%$. The bandwidth overhead in this case is 7.1%.

For a given virtual topology, we include all paths meeting the loss criteria; thus, most paths are as reliable as the IEEE standard requires. Even if coding overhead is high for a particular flow, the average overhead will be small. We show later (§7) that the number of good paths for which error correction is unnecessary dominates the total number of paths.

5.4 Configuring RAIL

While it is up to individual operators to configure the number and loss rate guarantee of virtual topologies in RAIL, based on their applications, we offer a simple recommendation. It is based on the double observation that many applications use TCP, and the performance of some TCP variants degrade noticeably only when loss rate exceeds 0.1% [48, 26]. We see this behavior in our experiments, and it is consistent with what others have reported. That is why some operators completely switch off links with error rates higher than 0.1% [58].

We recommend that data centers be configured with three virtual topologies. The first-class topology should provide paths with the same reliability as today, equivalent to where each link has BER less than 10^{-12}. This can be used to carry the most sensitive applications, such as RDMA. The second-class topology should provide paths with PPER below 0.1%, and it should carry applications like short TCP flows. Finally, the third-class topology should provide paths with PPER below 10%, and unless the application uses loss-tolerant transport, it should be error corrected. When the application is a long TCP flow, it should be error corrected to a reliability of t=0.1%. RAIL always put RDMA traffic on first-class topology and never uses error-correction code with RDMA. A corollary of our recommendation is that any link with PPER above 10% will be turned off entirely. Such links are rare (§7).

6 Implementation

Our implementation of the RAIL controller and the error correction module includes the following features. The controller learns link PERs from CRC error counters. It does not distinguish between optical- versus electrical-related corruption and provides protection from both. It constructs three virtual topologies with worst-case PPER of 0.0001%, 0.1% and 10%. The controller computes routing tables globally based on the virtual topologies and pushes rules to switches accordingly.

Our implementation of error correction sits below the kernel TCP/IP stack so that it is oblivious to the transport protocol. We implement it as a driver for tun/tap device in the Linux kernel. (On Windows, a WinSock kernel device driver may be used.) The driver keeps a buffer of size n so that it can decode the coded packet if needed and deliver packets to higher layers in order. It keeps a fine-grained timer such that if a missing packet is not recovered within a short time window, the next packet is delivered to the transport protocol (e.g., TCP). This delivery may trigger a recovery at the transport layer. Such transport-layer retransmissions are new packets for our error module.

To identify packets for coding and decoding, we insert a 4-byte header after the IP header containing a sequence number. Once the encoding rate is negotiated, coded packets and data packets have separate sequence numbers. The parity packet has the last sequence number in each group of n. As the error correction module knows the exact path to the destination, it performs cross flow error correction among all the flows with the same path.

7 Evaluation

Our evaluations are divided into three categories. First, we use a testbed and an optical simulator to quantify the stretch of two widely used short reach transceivers and compute the resulting BER and potential cost savings (§7.1). Next, we evaluate the impact of stretching these transceivers on the overall network path quality (§7.2). Finally, we show RAIL preserves applications' performance in a network with gray links (§7.3). Our results demonstrate that RAIL has minimal impact on overall network quality and application performance, while reducing total network cost by up to 10% for 10Gbps networks and up to 44% for 40Gbps networks.

7.1 Stretching Existing Technologies

We experiment with eight IEEE-standard short-reach 10Gbps and 40Gbps transceivers (two brands for each speed and two units of each brand). Some manufacturers provide non-standard technologies with reach values that are not supported by IEEE. These transceivers can likely be stretched as well if they follow similar specification practices, but evaluating individual non-standard transceivers is beyond the scope of this paper. Our goal rather is to demonstrate that commodity transceivers are over-engineered and can be stretched beyond their specification. We also experimented with a standard 100Gbps transceiver. Those results are preliminary and appear in Appendix C.

Experimental methodology. Our stretch experiments are based on a testbed and an optical simulator. Our testbed has one 10G [2] and one 40/100G switch [3], four 10G-SR transceivers [8, 4], four 40G-SR4 transceivers [9, 4] and a set of OM3 fibers [5] of lengths between 10m to 1000m. We focus on SR (short reach) technologies as they are viable candidates for stretching; the reach of LR (long reach) technologies is longer than typical maximum DC link lengths.

To emulate long fibers of different lengths, as shown in Figure 10, we concatenate multiple short cables with fiber connectors. To emulate additional attenuation in real environments, due to more/dirty connectors or damaged

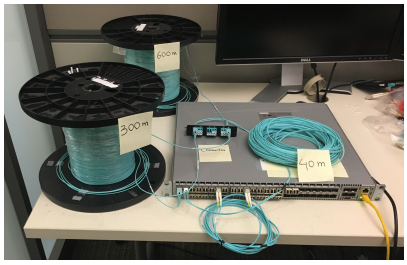
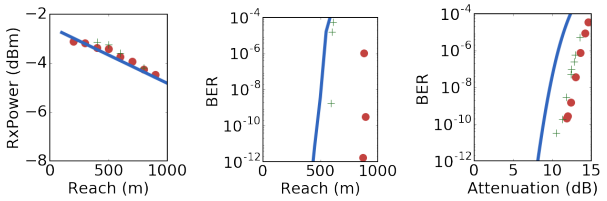


Figure 10: *Our 10G-SR testbed. We stitch short fibers together with fiber connectors to emulate long fibers. We concatenate 600m, 300m, 40m fibers together to emulate a 940m fiber. Two 300m 10G-SR Finisar transceivers [8] are attached to the fiber on both ends. Switch ports are on and we can send traffic through with BER of around* 10^{-8}.



(a) *RxPower vs. reach* (b) *BER vs. reach* (c) *Attenuation vs. BER*

Figure 11: *Validation for 40G-SR4 on OM3 fiber. Blue line corresponds to simulations. Circles and crosses are testbed results for transceivers from different manufacturers. (a) RxPower as a function of fiber length (no added attenuation). (b) BER as a function of fiber length (no added attenuation). (c) BER as a function of added attenuation when fiber is at transceiver's specified reach.*

fiber, we insert a variable attenuator which adds additional attenuation from the distribution shown in Figure 5a.

Since our transceivers do not directly report BER, to infer their BER, we measure packet corruption rate (LPER) with full-sized, line rate traffic for five minutes. We then calculate BER from LPER using a simplified model:[5] $LPER = 1-(1-BER)^{PACKET\ SIZE}$.

Testbed experiments are useful to provide coarse data on how link BER changes with different link lengths and attenuation levels. To explore the parameter space in fine granularity and to eliminate hardware quality differences between manufacturers, we use VPI [15], a standard optical simulator for data transmission system. (We cannot use a close-form formula to compute BER based on fiber length because of complex dispersion effects.) Our simulations model laser characteristics and a laser driver in the sender, modal and chromatic dispersion in the fiber, loss on the connector, and receiver sensitivity and dispersion equalization chips in the receiver. We configure these parameters based on the transceivers' specification sheet.[6]

For both 10Gbps and 40Gbps, we validate our simulator along three dimensions: RxPower, BER, and attenuation. Figure 11 shows our validation results for 40G-SR4; the results are similar for 10G-SR4. Figure 11a shows RxPower as fiber length increases. The scatter dots are testbed results, and the solid line is the result of our simulator. The figure shows that the simulator is able to closely match the RxPower of both transceiver brands for all fiber lengths. Figure 11b shows BER as fiber length increases. Since BER depends on a transceiver's sensitivity to modal dispersion, the two transceivers do not necessarily have to match. Hence, we configure our simulator to be the most sensitive one of the two brands and use the more conservative results in our evaluations for the rest of this paper. Similarly, Figure 11c shows that our simulations capture BER vs. additional attenuation conservatively. In this experiment, we use a 100m fiber, the design limit of our 40G-SR4 transceivers, and introduce additional attenuation using a variable attenuator.

BER distribution versus link length. We can now derive the distribution of BER that a link will observe as a function of its length. This distribution is a function of both link length and additional attenuation caused by dirty connectors or damaged fiber in real deployments. To simulate how stretched links impact BER in real world, we add the attenuation distribution seen in our measurement data in §2. This method is conservative because it assumes that all the attenuation measured in the wild is caused by factors other than link length. In practice, link length contributes as well, and our simulator already includes link length.

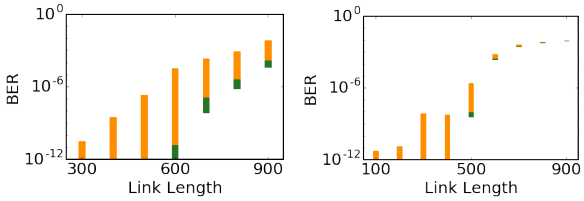
Figure 12 shows the BER distribution that will occur for various link lengths. The BER is represented as a bar for each link length. The top of the bar represents when extra attenuation is 99.9%-tile value (from the attenuation data) and the middle, which separates the two colors, when it is 99%-tile. As the figure shows, when we use 10G-SR, which is rated for 300m, on 500m links, at least 99% of these links will have BER less than 10^{-12}. At the same performance level, we can stretch 40G-SR4, which is rated for 100m, to 400m.

Cost savings. The level of stretch that a network should use depends on the trade-off between cost savings and performance, which we quantify using the NRB metric defined in §3. More stretch means more cost savings, but it also means that a larger fraction of paths is gray.

We illustrate this trade-off using a standard 3-stage Clos network. A DCN's total cost includes the equipment costs (i.e., transceivers, fibers, switches) and switch power consumption. Switches are $90/port [50], multi-mode and single-mode fiber cost $0.44 and $0.21 per meter respectively [6]. Each switch consumes around 150 Watts

[5] This model may overestimate BER, and this would underestimate potential for stretch because of two factors. First, some packet corruption events may be due to non-optical issues. Second, Ethernet uses line code (e.g., 64b/66b for 10Gbps network), and any bit flip in the code causes an extra 2 bits to be corrupted in the future, possibly in subsequent packets.

[6] Our simulation files are available online [17]; other researchers can use these to simulate optical links in DCNs.



(a) *10G-SR* (b) *40G-SR4*

Figure 12: *BER distribution for different link lengths. The top of the orange bar is 99.9 percentile BER, and the bottom of the green bar is 0 percentile BER. The two colors meet at 99% BER. The green portion is not visible when BER is below* 10^{-12}.

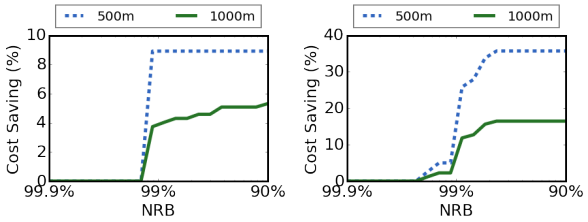
Device	Unit Cost	Count	Total Cost
300m 10G-SR Transceiver	$82.3	19660	$1.6M
10km 10G-LR Transceiver	$187.2	13108	$2.5M
Multi-mode Fiber (10Gbps)	$0.44/m	1475 km	$0.6M
Single-mode Fiber (10Gbps)	$0.21/m	2621 km	$0.6M
Switch (10Gbps)	$90/port	32768 ports	$2.9M
Power(150W/Switch, 3 years)	$0.07/KWh	5 GWh	$0.4M

Table 1: *Total DCN cost breakdown for 512 ToR, 512 aggregation and 256 core switches Clos network with 10Gbps technology. Link length is drawn from uniform distribution between 0-500m. With 10Gbps technology, transceivers account for 48% of the total DCN cost.*

(this includes energy consumed by transceivers). With 32-port switches, we can build a full bisection 3-stage fat tree network with 512 ToR, 512 aggregation, and 256 core switches. There are 8192 ToR-aggregation links and 8192 aggregation-core links. The link length distribution depends on the physical layout of the DCN. For links under 300m, 10G-SR is used with multi-mode fiber. For links above 300m, 10G-LR is used with single-mode fiber. Table 1 provides a breakdown of the cost of these DCNs. As the table shows, transceivers represent 48% of the total cost of the network. If the same network were using a 300m 10GBase-SR transceiver stretched to 500m (for example), all the 300–500m links could use this transceiver instead of the more expensive 10GBase-LR one.

Figure 13 shows the cost reduction versus NRB for 0-500m uniform and 0-1000m uniform link length distributions. For these plots, we compute the stretch level from NRB as outlined in §3 and then the cost savings from the stretch level. Our measurement data (§2) show NRB in current networks is 99.9%. We see that cost savings are significant when NRB drops to 99%, and the incremental gain is small beyond 95%. Thus, in later evaluations, we only consider two degrees of stretch, low stretch for NRB=99% and high stretch for NRB=95%, which correspond to stretch levels in Table 2.

Next, we study how link length distribution affects the amount of cost savings. Figure 14a shows the resulting cost savings for a 10Gbps network. When no link is longer than 300m, stretch does not result in any savings because we



(a) *10Gbps* (b) *40Gbps*

Figure 13: *Total DCN cost reduction on a 3-stage fat tree network (10Gbps, 40Gbps) with 512 ToRs assuming 0-500m and 0-1000m uniform link length distribution. The cost reduction depends on the amount of stretch.*

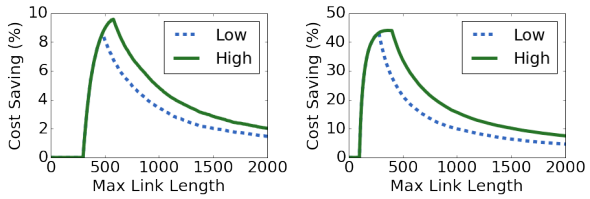
Technology	No Stretch	Low	High
10G-SR (OM3)	300m	480m	580m
40G-SR4 (OM3)	100m	280m	400m

Table 2: *Optical technologies' maximum reach under different degrees of stretch.*

never use stretched technologies. The cost saving peaks at 480m for low stretch and 580m for high stretch, the lengths at which most fractions of links can use shorter reach technologies beyond their design reach. When link length distribution concentrates on long links, the amount of cost savings decreases because most of the links need to use long reach technologies anyway, and cost on long reach technologies dominates the total cost of the network.[7] Overall, stretching short reach technology reduces the total cost DCN-wide up to 10% (1 million dollars) depending on the link length distribution. The savings are lower for low stretch, because fewer 10GBase-LR transceivers can be changed to 10GBase-SR transceivers.

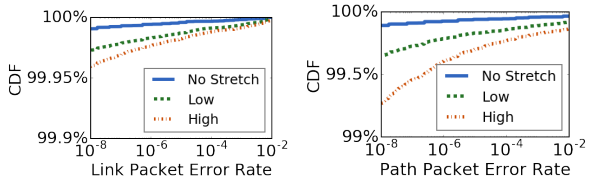
Figure 14b shows the cost results for 40Gbps networks. Except for transceiver costs, which are listed in Figure 7, we assume the cost and energy consumption of 40Gbps components is 4× higher than their 10Gbps counterparts—40Gbps components often bundle four 10Gbps components. While we assume multi-mode fiber is $1.32 per meter, we keep the cost of single mode fiber the same because 40Gbps single-mode transceivers use wavelength division multiplexing. Transceivers account for 72% of the total DCN cost for 0-500m uniform link length distribution. The overall trend of cost savings is similar to that in the 10Gbps network. The total cost savings on 40Gbps networks can be up to 44% (21 million dollars) depending on the link length distribution. The cost reductions are higher for these networks because *i*) there is a higher cost difference between short and long reach technologies, *ii*) a larger fraction of the links can make use of shorter

[7] If the network uses non-standard, intermediate reach (e.g., 500m) transceivers, we could have stretched them to cover longer distances (e.g., 500m to 1km) as well.



(a) *10Gbps* (b) *40Gbps*

Figure 14: *Total DCN cost reduction on a 3-stage fat tree network (10Gbps, 40Gbps) with 512 ToRs assuming uniform link length distribution. Maximum cost savings for 10/40Gbps is achieved when max link length is 580/400m.*



(a) *LPER* (b) *PPER*

Figure 15: *(a) CDF of LPER. (b) CDF of PPER when oversubscription is 4.*

reach technology, and *iii*) transceivers account for a larger fraction of total DCN cost based on our cost assumptions.

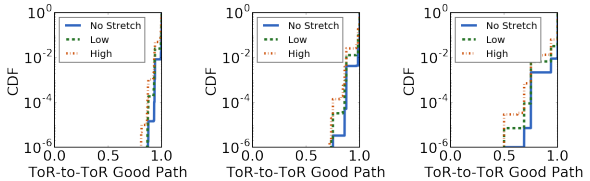
7.2 Characterizing a Stretched Network

NRB ensures by design that the network has a certain minimum fraction of good paths. We now provide a more detailed characterization of a stretched network in terms of the distribution of its link and path qualities. These distributions depend on the exact link length distribution in the network. So that our results can be reproduced, we use 0-500m uniform link length distribution on a 512-ToR Clos network (as in §7.1). Results with link lengths drawn from our cloud provider's network were similar. We study 40Gbps below; 10Gbps networks behave similarly.

Link qualities. Figure 15a shows the CDF of LPERs with and without stretch. We see that even with high stretch, only 0.05% of the links have LPER worse than 10^{-8} (which corresponds to 10^{-12} BER). Only 0.01% of the links have LPER higher than 10%, our guideline for switching off links. As reference, we note that at any given time, roughly 0.08% of the links are down for other reasons in our data centers.

Path qualities. To study path characteristics in a stretched network, we simulate three different oversubscription levels. When the oversubscription level is 1 (4, 16), we have 512 (256, 128) aggregation switches and 256 (128, 64) core switches. Each switch has 32 ports.

We assume the links in the topology have LPERs as per the distribution above. We showed earlier that low RxPower (and thus high LPER) levels are not correlated with switches and appear independent across links. For



(a) *Oversub = 1* (b) *Oversub = 4* (c) *Oversub = 16*

Figure 16: *CDFs of the fraction of good paths for different ToR-ToR pairs across 50 different topologies. The Y axis is log scale to capture the tail behavior.*

each oversubscription level, we generate multiple topologies with different randomized mappings from the LPER distribution and present results aggregated across them.

Figure 15b shows the CDF of PPER for a 512-ToR topology with an oversubscription of four. The results are similar for other oversubscription levels; 99% of paths in the topology have PPER below 10^{-8}.

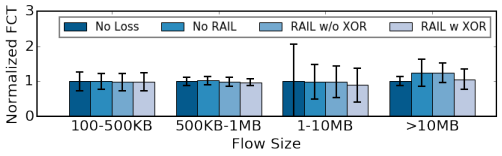
Worst-case experience of loss-sensitive applications. An overall high fraction of good paths is not sufficient to ensure good performance for loss-sensitive applications. For every pair of ToRs that exchange traffic for such applications, there must be enough good paths. Figure 16 shows the CDF of the ratio of good paths to total paths across ToR pairs. A good path means PPER less than 4.8×10^{-8}, equivalent to 10^{-12} BER on each link of a path in a 3-stage fat tree.

We see that when the oversubscription is equal to one, the tail 0.01% of the ToR-to-ToR pairs still has 83% of the good paths remaining. This fraction decreases as oversubscription increases because ToR switches now have fewer uplinks to aggregation switches. If one such uplink has high LPER, it impacts a higher fraction of paths from this ToR to other ToRs. However, even when the oversubscription is 16, the tail 0.01% of the ToR-to-ToR paths still has 70% of good paths left. On the flip side, these data also demonstrate that simply turning gray links off can halve the capacity between some ToR pairs.

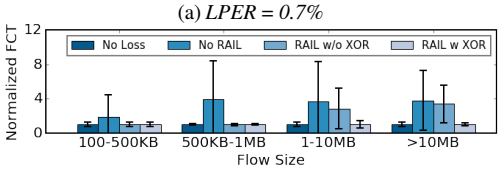
7.3 Application Performance with RAIL

We study the effectiveness of RAIL in preserving application performance using experiments on a small but realistic testbed. Our testbed emulates a 3-stage fat tree network with four ToRs, four aggregation switches, and two core switches. The ToR and aggregation switches are spread across two pods. Each port in the topology is a 10Gbps SPF+ port, and each link has two Finisar FTLX8571D3BCL [8] multi-mode transceivers connected via OM3 fiber [5]. The switches implement ECMP routing. One host is attached to each ToR switch, runs Ubuntu 14.04 with Linux kernel version 3.19, and uses TCP CUBIC [36].

We emulate a gray, high-BER link using an optical attenuator and change the location of the attenuator in different experiments. We use the virtual topology configuration



(a) *LPER = 0.7%*



(b) *LPER = 5%*

Figure 17: *Normalized flow completion time. Error bars are standard deviations of the normalized flow completion time. Flows shorter than 1MB are high priority flows. High priority flows are not affected by packet corruption with RAIL. Low priority flows are protected by XOR coding.*

guideline above, but because it is difficult to finely control the BER using an attenuator, we obtain configurations in which the first- and second-class topologies are identical and contain highly-reliable paths. The third-class topology contains the high-BER link(s). We use iperf to send TCP flows between arbitrary end hosts, with a flow size and inter-arrival time distribution from prior work [20]. TCP flows smaller than 1MB bind to the second- class topology; longer flows bind to the third-class topology.[8]

Figure 17 shows flow completion times (FCT) binned by flow size and normalized to the case of a completely lossless network ("No Loss"). When the network is lossy, without RAIL ("No RAIL"), we see that FCTs are higher for all flow sizes, especially when the loss rate is high. RAIL without error coding ("RAIL w/o XOR") is able to protect only high-priority (short) flows because it routes them over the reliable, second-class topology. With error coding, RAIL ("RAIL w XOR") is able to protect low-priority flows as well. The performance experienced by all flows matches that of the lossless network.

Error correction overhead. Finally, we study the bandwidth overhead of RAIL's error correction. Recall that we set the target post-recovery loss rate to 0.1%. Thus, error correction only kicks in for paths with PER above 0.1%. For paths with PER between 0.1% and 3%, we use standard XOR code with n computed based on error rate. For PPER from 3–10%, we simply replicate every packet three times.

The average bandwidth overhead of our coding scheme is below 0.1% for the 512-ToR topology with 0-500m and 0-1000m uniform link length distribution. This low overhead is the reason we use a simple coding method in RAIL instead of more efficient methods based on retransmissions or bit-level FEC (forward error correction).

[8]This mapping between flow size and topology is only for our experiments. In reality, we expect applications to bind to the desired virtual interface. RAIL does not try to guess flow sizes.

8 Related Work

Our work draws on several themes of previous work.

Measuring optical links. Many researchers have studied optical WAN for properties such as dispersion [30, 57, 25, 39, 56], temperature variations [38, 42], and packet loss and inter-arrival times [31, 43]. Ghobadi et al. study optical signal quality in WAN and, like us, find an overprovisioned optical layer [33]. In contrast, however, our focus is on DCNs, where the environment and technology are different. We believe we are the first to study this optical layer.

Reducing cost of optics in DCNs. We are inspired by other efforts in the industry to lower the cost of optics in DCNs. Facebook [27] is pushing for a new standard for low cost 100Gbps transceivers. Their initial observation is similar to ours: the DCN is a milder operating environment than traditional telecom networks. Corning [28] also observed, using stochastic attenuation models, that DCN link qualities can be disparate and it is possible to extend the reach on some fraction of links. We complement these efforts with a detailed characterization of optical links in operational DCNs, proposing a way to reduce cost without hardware changes and developing a system to preserve application performance if some links turn gray due to reduced over-engineering.

Virtual topologies. The concept of virtual topologies over the same physical infrastructure has been leveraged in other contexts, such as simplifying the specification of network policies in software-defined networking (SDN) [40, 51, 46] or "slicing" the network to isolate users [23, 54, 47]. We use this concept to build topologies with different reliability guarantees.

Reliable systems atop unreliable components. There is a long-standing tradition of building reliable systems using (cheaper) unreliable components and masking unreliability from applications using intelligent software techniques. Classic examples include building reliable storage systems using disks that are individually unreliable [49, 32]. Our work follows this tradition, though the set of techniques it uses are specific to its domain.

9 Conclusions

Our measurements show that optical links in data center networks are heavily over-engineered. This over-engineering is not only expensive but also unnecessary because of application and path diversity in DCNs. Many applications can tolerate small amounts of loss, and loss-sensitive applications can be supported as long as some (not all) paths between ToR pairs are reliable. We find that reducing optical over-engineering simply by using transceivers beyond their specified length can reduce network cost by up to 10% for 10Gbps networks and 44% for 40Gbps networks. Moreover, when coupled with the traffic routing and error correction mechanisms of RAIL, there is negligible loss in application performance.

Acknowledgments

We thank Keren Bergman, David Bragg and Jamie Gaudette for sharing their expertise on optics and Hui Ma and Shikhar Suri for helping with the deployment of our monitoring system. We also thank our shepherd George Porter and the anonymous reviewers for their feedback on the paper. This work was partially supported by the NSF (CNS-1318396 and CNS-1518702).

References

[1] AOI 100G MPO MMF 850nm 70m Transceiver. http://www.high-tech.co.jp/common/sys/product/product00445_01.pdf.

[2] Arista 7050S. https://www.arista.com/assets/data/pdf/Datasheets/7050S_Datasheet.pdf.

[3] Arista 7060CX. https://www.arista.com/assets/data/pdf/Datasheets/7060X_7260X_DS.pdf.

[4] Arista Optics Modules and Cables. https://www.arista.com/assets/data/pdf/Datasheets/arista_transceiver_datasheet.pdf.

[5] Corning OM3 Fiber. https://www.corning.com/media/worldwide/coc/documents/Fiber/PI1468_07-14_English.pdf.

[6] Duplex Zipcord Fiber Optic Cable. http://www.fs.com/c/duplex-zipcord-fiber-optic-cable-1249.

[7] Finisar. http://www.mouser.com/finisar-corporation/.

[8] FINISAR FTLX8571D3BCL. https://www.finisar.com/optical-transceivers/ftlx8571d3bcl.

[9] FS.COM QSFP 40G Transceiver. http://www.fs.com/products/36309.html.

[10] HPC Optics. https://www.hpcoptics.com.

[11] IEEE 10 Gb/s Ethernet task force. http://grouper.ieee.org/groups/802/3/ae/public/blue_book.pdf.

[12] Optical monitor MIB. http://www.oidview.com/mibs/9/CISCO-OPTICAL-MONITOR-MIB.html.

[13] PCWholeSale. http://www.pc-wholesale.com/.

[14] ROBOfiber. http://www.robofiber.com/.

[15] VPI. http://www.vpiphotonics.com/index.php.

[16] 100G PSM4 Specification. http://www.psm4.org/100G-PSM4-Specification-2.0.pdf, 2014.

[17] RAIL VPI simulation files. https://github.com/railnsdi2017/rail_VPI, 2016.

[18] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[19] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *SIGCOMM*, 2014.

[20] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *SIGCOMM*, 2013.

[21] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *SOSP*, 2009.

[22] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz. A Comparison of Mechanisms for Improving TCP Performance over Wireless Links. In *SIGCOMM*, 1996.

[23] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In *SIGCOMM*, 2006.

[24] E. Brewer, L. Ying, L. Greenfield, R. Cypher, and T. Ts'o. Disks for Data Centers. Technical report, Google, 2016.

[25] H. Bulow, W. Baumert, H. Schmuck, F. Mohr, T. Schulz, F. Kuppers, and W. Weiershausen. Measurement of the maximum speed of PMD fluctuation in installed field fiber. In *OFC*, 1999.

[26] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. BBR: Congestion-Based Congestion Control. *ACM Queue*, 2016.

[27] A. Chakravarty, K. Schmidtke, S. Giridharan, J. Huang, and V. Zeng. 100G CWDM4 SMF Optical Interconnects for Facebook Data Centers. *Conference on Lasers and Electro-Optics*, 2016.

[28] X. Chen, J. Abbott, D. Powers, D. Coleman, and M.-J. Li. Statistical Treatment of IEEE spreadsheet model for VCSEL-multimode fiber transmissions. *OECC/PS*, 2016.

[29] N. El-Sayed, I. A. Stefanovici, G. Amvrosiadis, A. A. Hwang, and B. Schroeder. Temperature Management in Data Centers: Why Some (Might) Like It Hot. *SIGMETRICS Perform. Eval. Rev.*, 2012.

[30] R. J. Feuerstein. Field Measurements of Deployed Fiber. In *OFC*, 2005.

[31] D. A. Freedman, T. Marian, J. H. Lee, K. Birman, H. Weatherspoon, and C. Xu. Exact Temporal Characterization of 10 Gbps Optical Wide-area Network. In *IMC*, 2010.

[32] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SOSP*, 2003.

[33] M. Ghobadi, J. Gaudette, R. Mahajan, A. Phanishayee, D. Kilper, and B. Klinkers. Signal characteristics in Microsoft optical backbone. In *OFC*, 2016.

[34] P. Gill, N. Jain, and N. Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *SIGCOMM*, 2011.

[35] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[36] S. Ha, I. Rhee, and L. Xu. CUBIC: A New TCP-friendly High-speed TCP Variant. *SIGOPS Oper. Syst. Rev.*, 2008.

[37] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: Saving Energy in Data Center Networks. In *NSDI*, 2010.

[38] H. Ji, J. H. Lee, and Y. C. Chung. System Outage Probability Due to Dispersion Variation Caused by Seasonal and Regional Temperature Variations. In *OFC*, 2005.

[39] M. Karlsson, J. Brentel, and P. Andrekson. Long-term measurement of PMD and polarization drift in installed fibers. *Journal of Lightwave Technology*, 2000.

[40] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network Virtualization in Multi-tenant Datacenters. In *NSDI*, 2014.

[41] P. Lahiri, G. Chen, P. Lapukhov, E. Nkposong, D. Maltz, R. Toomey, and L. Yuan. Routing Design for Large Scale Data Centers: BGP is a better IGP! https://www.nanog.org/meetings/nanog55/presentations/Monday/Lapukhov.pdf.

[42] J. C. Li, K. Hinton, P. M. Farrell, and S. D. Dods. Optical impairment outage computation. *Opt. Express*, 2008.

[43] T. Marian, D. Freedman, K. Birman, and H. Weatherspoon. Empirical characterization of uncongested optical lambda networks and 10GbE commodity endpoints. In *DSN*, 2010.

[44] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang. TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links. In *MobiCom*, 2001.

[45] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 2008.

[46] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-defined Networks. In *NSDI*, 2013.

[47] J. Mudigonda, P. Yalagandula, J. Mogul, B. Stiekes, and Y. Pouffary. NetLord: A Scalable Multi-tenant Network Architecture for Virtualized Datacenters. In *SIGCOMM*, 2011.

[48] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Throughput: A Simple Model and Its Empirical Validation. In *SIGCOMM*, 1998.

[49] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD*, 1988.

[50] L. Popa, S. Ratnasamy, G. Iannaccone, A. Krishnamurthy, and I. Stoica. A Cost Comparison of Datacenter Network Architectures. In *CoNEXT*, 2010.

[51] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *SIGCOMM*, 2013.

[52] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *SIGCOMM*, 2011.

[53] L. Rizzo. Effective Erasure Codes for Reliable Computer Communication Protocols. *SIGCOMM Comput. Commun. Rev.*, 1997.

[54] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the Production Network Be the Testbed? In *OSDI*, 2010.

[55] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM*, 2015.

[56] S. Woodward, L. Nelson, M. Feuer, X. Zhou, P. Magill, S. Foo, D. Hanson, H. Sun, M. Moyer, and M. O'Sullivan. Characterization of Real-Time PMD and Chromatic Dispersion Monitoring in a High-PMD 46-Gb/s Transmission System. *Photonics Technology Letters, IEEE*, 2008.

[57] S. Woodward, L. Nelson, C. Schneider, L. Knox, M. O'Sullivan, C. Laperle, M. Moyer, and S. Foo. Long-Term Observation of PMD and SOP on Installed Fiber Routes. *Photonics Technology Letters, IEEE*, 2014.

[58] X. Wu, D. Turner, G. Chen, D. Maltz, X. Yang, L. Yuan, and M. Zhang. NetPilot: Automating Datacenter Network Failure Mitigation. In *SIGCOMM*, 2012.

[59] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *EuroSys*, 2014.

[60] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM*, 2015.

Appendix

A Expected Fraction of Good Paths

Definition 2. *A path is good when all unidirectional links on the path are good.*

Lemma 1. *For a path with h hops, when each hop has the probability of at least l to be good and independent, the path is good with the probability of at least l^h.*

Proof. Every hop's quality is independent. □

Theorem 1. *When every path is good with the probability of at least P_{each}, the expected fraction of good paths is at least P_{each}.*

Proof. Let P_i be the probability of a path i to be good. Let's define an indication variable I_p such that

$$I_p = \begin{cases} 0 \text{ , if path p is not good,} \\ 1 \text{ , if path p is good} \end{cases}$$

If the network has n paths, the expected fraction of paths is

$$\text{Expected fraction of good paths} = E\left(\frac{\sum_{p \in \text{all paths}} I_p}{n}\right)$$

Using the linearity of expected value, we get

$$\text{Expected fraction of good paths} = \frac{\sum_{p \in \text{all paths}} E(I_p)}{n}$$

Because $E(I_p) \geq P_{each}$,

$$\text{Expected fraction of good paths} \geq \frac{nP_{each}}{n} = P_{each}$$

□

Theorem 2. *If each link has the probability of at least l to be good and independent, and the longest path in the network has h hops, then the expected fraction of good paths is at least l^h.*

Proof. Use Lemma 1 to get a lower bound of the probability of any path to be good. Then use Theorem 1. □

B Find the Worst Path in Clos Topology

We seek an efficient solution to the following problem: Assuming ECMP routing, given a subset of links on a Clos network and every link's unidirectional packet error rate (LPER), find the worst end-to-end path with the highest path packet error rate (PPER).

This problem can be solved efficiently because of the data center's unique topology and its simple ECMP routing scheme. We describe our algorithms assuming the Clos topology has an oversubscription ratio of 1 across all stages. All our results hold when oversubscription is introduced.

Definition 3. *A Clos network is a multi-stage network. A switch at (i)th stage can only connect to switches at neighbor stages (i.e., $(i \pm 1)$th stage). Every stage has n switches, except for the top stage which has $\frac{n}{2}$ switches. All switches in the network have $2k$ ports. The network has $\log_k n$ stages. Every switch except on the top stage has k ports facing the upper stage and k ports facing the lower stage. Switches on the top stage have all $2k$ ports facing the lower stage.*

Today, data center operators chose ECMP up-down routing as the basic routing algorithm. In this method, packets first travel to an upper stage switch and then down to the destination top-of-rack (ToR) switch. ECMP requires packets to take one of the shortest paths.

Definition 4. *A path from ToRs (stage = 1) switches a to b ($a \neq b$) is called an up-down path when there is a switch c on the path such that the stage number of each switch monotonically increases from a to c and monotonically decreases from c to b.*

Clos topology has a unique property, in that every up-down path can actually be chosen to forward traffic. All up-down paths between the same ToR-pair have exactly the same number of hops. When we want to find the worst path, we only have to consider all the up-down paths.

The worst path is defined as the following.

Definition 5. *A worst path is the up-down path with highest PPER. PPER of path p is* $1-\prod_{l\in p}(1-LPER_l)$.

It is computationally infeasible to track PPER for every path in the network. Tracking PPER takes at least $O(n^3)$ running time because a Clos network has $O(n^3)$ paths (i.e., $O(n)$ paths for every ToR pair and there are $O(n^2)$ ToR pairs). As an alternative, the worst path computation must quickly neglect paths with no hope of becoming the worst path in the run-time.

Our algorithm 1 identifies the worst path with a running time of $O(n\log n)$ and is provably optimal in asymptotic running-time. In this section, we prove the path produced by the algorithm is indeed the worst path; we then prove no faster algorithm exists.

Lemma 2. *The output of Algorithm 1 is an up-down path.*

Proof. We only need to show $\forall s$,top_s is a valid path. From Algorithm 1, top_s is nothing but a monotonically up-going path to s and a monotonically downward path from s.

Thus, top_s is an up-down path if the source and the destination ToR of top_s are different. In Algorithm 1, we see when we calculate top_s, we enforce the direct children of s to be different. In Clos network, this means the source and the destination ToR are in different branches of the Clos (subtree of fat tree) containing non-overlapping sets of ToRs. Thus, top_s is an up-down path. □

Theorem 3. *The output of our algorithm is the worst path.*

Proof. Proof by contradiction. Our algorithm outputs path p. The worst path is p' such that $PPER_{p'} > PPER_p$.

Without loss of generality, p' is an up-down path from a' to b'. Because p' is also an up-down path, by definition, there must be a c' on p' such that c' is the highest stage switch on p', a' to c' is a monotonic upward path, and c' to b' is a monotonic downward path.

When our algorithm goes through switch c', there are two situations. (1) $p' = top_{s'}$ (2) $p' \neq top_{s'}$. Let's talk about both situations.

If $p' = top_{s'}$, because $PPER_p = max_{s\in all\ switches}(top_s)$, $PPER_p \geq top'_s = p'$. This contradicts the assumption that $PPER_{p'} > PPER_p$.

If $p' \neq top_{s'}$ and p' is valid, p' must includes two children of s'. Then, it must be the case that $p' < top_{s'}$ because, otherwise, p' will be chosen when computing $top_{s'}$. Because $PPER_p = max_{s\in all\ switches}(top_s)$, then $PPER_p \geq top_{s'} > p'$. This contradicts the assumption that $PPER_{p'} > PPER_p$.

Overall, $PPER_{p'} \leq PPER_p$. Thus, the output of our algorithm is the worst path.

□

Theorem 4. *The running time of our algorithm is* $\Theta(n\log n)$.

Proof. Our algorithm passes through each switch once, and the update algorithm is $\Theta(k^2)$. Because k is constant, the update part is in the order of number of switches, $\Theta(n\log_k n)$. The comparison part compares top_s for all switches which takes another $\Theta(n\log_k n)$. Thus, our algorithm finishes in $\Theta(nlog_k n)$. Because k is constant, our algorithm finishes in $\Theta(n\log n)$. □

Theorem 5. *Any algorithm that can compute the worst path has a running time of at least* $\Theta(n\log n)$.

Proof. Proof by contradiction. Assume an algorithm exists that computes the worst path with running-time less than $\Theta(n\log n)$.

We construct an adversarial oracle that always returns LPER = 0.5 for every link queried by the algorithm. After the algorithm finishes, the algorithm returns a worst path p. Because the number of links is in the order of $\Theta(nlog_k n)$, the algorithm does not have enough time to read LPER on every link. There must be a fraction of links that are not read by the algorithm. Let l be one of the unidirectional links whose LPER is not read by the algorithm.

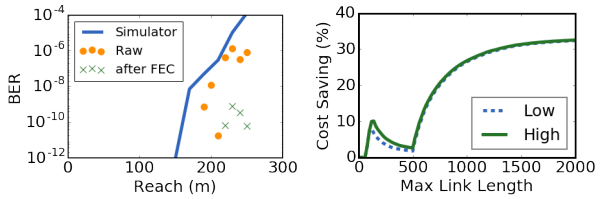
There are two situations here. (1) l is on p; (2) l is not on p. We discuss both situations.

If l is on p, the oracle sets $LPER_l \leftarrow 0$ and LPER of all remaining links (those not read by the algorithm) at 0.5. Thus, l cannot be the worst path because it has lower PPER than other paths. This contradicts the assumption that p is the worst path.

If l is not on p, the oracle sets $LPER_l \leftarrow 1$ and LPER of all remaining links (those not read by the algorithm) at 0.5. Thus, l must be on the worst path of any path going through l has higher PPER (PPER = 1) than any path that does not go through l. This contradicts the assumption that p is the worst path.

Overall, the algorithm cannot output the correct worst path without taking at least $\Theta(n\log n)$.

□



(a) *Reach Test* (b) *Cost Saving*

Figure 18: *18a shows the reach test for 100G-SR4. Blue line is simulated transceiver from VPI. Yellow dots are raw BER and green dots are corrected BER. 18b shows cost saving for uniform random link length distribution for different max length.*

C Over-engineering in 100Gbps

We repeat our stretch experiments using 100G-SR4 transceivers [1] (standard reach 70m) and observe similar levels of over-engineering. However, an important distinction between 100Gbps and 10/40Gbps technologies is the presence of a Forward Error Correction (FEC) module in 100Gbps switches. This means the pre-FEC BER requirement is reduced from 10^{-12} to 5×10^{-5} and the FEC module boosts BER back to the standard on every hop. Figure 18a shows pre- and post- FEC BER as we stretch the fiber length without additional attenuation. Interestingly, even pre-FEC BER (Raw) is lower than 10^{-12} at 180m; this is 2.5 times higher than the standard reach, once again confirming the degree of over-engineering. We simulate this transceiver in VPI and conservatively bound the stretch to 110m and 130m to achieve *network reliability bounds* of 99% and 95% respectively where FEC is off.

From simulation in VPI, we show PSM4 technology can be stretched from 500m to 2km. Figure 18b shows the cost saving for uniform random link length distribution. Before 70m, there is no cost saving because all links are covered by 100G-SR4. The cost saving peaks at 150m, the stretched reach, because the largest fraction of the link can use 100G-SR4. Cost saving decreases at this point, because 500m PSM4 technology's cost is cheap. Cost saving increases after 500m because PSM4 technologies can replace higher cost 100G-LR4 technologies between 500-2km. Overall, we find the savings in 100G can be up to 30%.

When FEC is turned on, the link quality distribution among all the links is narrower. The amount of over-engineering remains the same, which means stretching transceivers can still reduce the cost of DCN. However, switching gray links off is likely enough to protect the network. We leave this problem for future investigation.

Enabling Wide-spread Communications on Optical Fabric with MegaSwitch

Li Chen[1] Kai Chen[1] Zhonghua Zhu[2] Minlan Yu[3]
George Porter[4] Chunming Qiao[5] Shan Zhong[6]
[1]*SING Lab@HKUST* [2]*Omnisensing Photonics* [3]*Yale* [4]*UCSD* [5]*SUNY Buffalo* [6]*CoAdna*

Abstract

Existing wired optical interconnects face a challenge of supporting wide-spread communications in production clusters. Initial proposals are constrained, as they only support connections among a small number of racks (e.g., 2 or 4) at a time, with switching time of milliseconds. Recent efforts on reducing optical circuit reconfiguration time to microseconds partially mitigate this problem by rapidly time-sharing optical circuits across more nodes, but are still limited by the total number of parallel circuits available simultaneously.

In this paper, we seek an optical interconnect that can enable unconstrained communications within a computing cluster of thousands of servers. We present MegaSwitch, a multi-fiber ring optical fabric that exploits space division multiplexing across multiple fibers to deliver rearrangeably non-blocking communications to 30+ racks and 6000+ servers. We have implemented a 5-rack 40-server MegaSwitch prototype with commercial optical devices, and used testbed experiments as well as large-scale simulations to explore MegaSwitch's architectural benefits and tradeoffs.

1 Introduction

Due to its advantages over traditional electrical networks at high link speeds, optical circuit switching technology has been widely investigated for datacenter networks (DCN) to reduce cost, power consumption, and wiring complexity [38]. Many of the initial optical DCN proposals build on the assumption of substantial traffic "concentration". Using optical circuit switches that only support one-to-one connections between ports, they service highly concentrated traffic among a small number of racks at a time (e.g., 2 or 4 [6, 50]). However, evolving traffic characteristics in production DCNs pose new challenges:

- **Wide-spread communications:** Recent analysis of Facebook cluster traffic reveals that servers can concurrently communicate with hundreds of hosts, with the majority of traffic to tens of racks [42]; and traces from Google [46] and Microsoft [15, 17] also exhibit such high fan-in/out wide-spread communications. The driving forces behind such wide-spread patterns are multifold, such as data shuffling, load balancing, data spreading for fault-tolerance, etc. [42, 46].

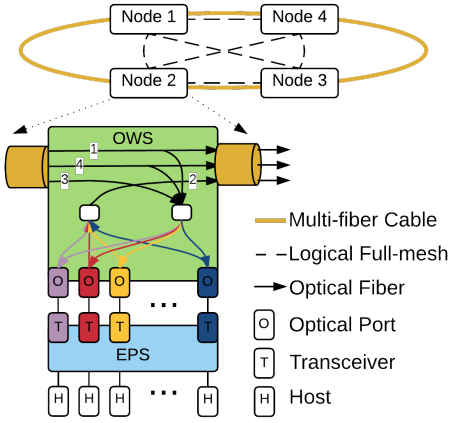


Figure 1: High-level view of a 4-node MegaSwitch.

- **High bandwidth demands:** Due to the ever-larger parallel data processing, remote storage access, web services, etc., traffic in Google's DCN has increased by 50x in the last 6 years, doubling every year [46].

Under such conditions, early optical DCN solutions fall short due to their constrained communication support [6, 7, 12, 50]. Recent efforts [24, 38] have proposed a *temporal* approach—by reducing the circuit switching time from milliseconds to microseconds, they are able to rapidly time-share optical circuits across more nodes in a shorter time. While such temporal approach mitigates the problem, it is still insufficient for the wide-spread communications, as it is limited by the total number of parallel circuits available at the same time [38].[1]

We take a *spatial* approach to address the challenge. To support the wide-spread communications, we seek a solution that can directly deliver parallel circuits to many nodes simultaneously. In this paper, we present an optical DCN interconnect for thousands of servers, called MegaSwitch, that enables unconstrained communications among all the servers.

At a high-level (Figure 1), MegaSwitch is a circuit-switched backplane physically composed of multiple optical wavelength switches (OWS, a switch we implemented §4) connected in a multi-fiber ring. Each OWS is attached to an electrical packet switch (EPS), forming a MegaSwitch node. OWS acts as an access point for EPS to the ring. Each sender uses a dedicated fiber to

[1] Another temporal approach [15] employs wireless free space optics (FSO) technology, However, FSO is not yet production-ready for DCNs, because dust and vibration that are common in DCNs can impair FSO link stability [15].

send traffic to any other nodes; on this multi-fiber ring, each receiver can receive traffic from all the senders (one per fiber) simultaneously. Essentially, MegaSwitch establishes a one-hop circuit between any pair of EPSes, forming a rearrangeably non-blocking circuit mesh over the physical multi-fiber ring.

Specifically, MegaSwitch achieves unconstrained connections by re-purposing wavelength selective switch (WSS, a key component of OWS) as a receiving $w\times1$ multiplexer to enable non-blocking space division multiplexing among w fibers (§3.1). Prior work [6, 11, 38] used WSS as $1\times w$ demultiplexer that takes 1 input fiber with k wavelengths, and outputs any subset of the k wavelengths to w output fibers. In contrast, MegaSwitch reverses the use of WSS. Each node uses k wavelengths[2] on a fiber for sending; for receiving, each node leverages WSS to intercept all $k\times w$ wavelengths from w other senders (one per fiber) via its w input ports. Then, with a wavelength assignment algorithm (§3.2), the WSS can always select, out of the $k\times w$ wavelengths, a non-interfering set to satisfy any communication demands among nodes.

As a result, MegaSwitch delivers rearrangeably non-blocking communications to $n\times k$ ports, where $n{=}w{+}1$ is the number of nodes/racks on the ring, and k is the number of ports per node. With current technology, w can be up to 32 and k up to 192, thus MegaSwitch can support up to 33 racks and 6336 hosts. In the future, MegaSwitch can scale beyond 10^5 ports with AWGR (array waveguide grating router) technology and multi-dimensional expansion (§3.1).

On top of its unconstrained connections, MegaSwitch further has built-in fault-tolerance to handle various failures such as OWS, cable, and link failures. We develop necessary redundancy and mechanisms, so that MegaSwitch can provide reliable services (§3.3).

We have implemented a small-scale MegaSwitch prototype (§4) with 5 nodes, and each node has 8 optical ports transmitting on 8 unique wavelengths within 190.5THz to 193.5THz at 200GHz channel spacing. This prototype enables arbitrary communications among 5 racks and 40 hosts. Furthermore, our OWSes are designed and implemented with all commercially available optical devices, and the EPSes we used are Broadcom Pronto-3922 10G commodity switches.

MegaSwitch's reconfiguration delay hinges on WSS, and 11.5μs WSS switching time has been reported using digital light processing (DLP) technology [38]. However, our implementation experience reveals that, as WSS port count increases (for wide-spread communications), such low latency can no longer be maintained. This is mainly because the port count of WSS with DLP used in [38] is not scalable. In our prototype, the WSS reconfiguration is $\sim3ms$. We believe this is a hard limitation we have to confront in order to scale. To accommodate unstable traffic and latency-critical applications, we develop "basemesh" on MegaSwitch to ensure any two nodes are always connected via certain wavelengths during reconfigurations (§3.2.2). Especially, we construct the basemesh with the well-known Symphony [29] topology in distributed hash table (DHT) literature to provide low average latency with adjustable capacity.

Over the prototype testbed, we conducted basic measurements of throughput and latency, and deployed real applications, e.g., Spark [54] and Redis [43], to evaluate the performance of MegaSwitch (§5.1). Our experiments show that the latency-sensitive Redis experiences uniformly low latency for cross-rack queries due to the basemesh, and the performance of Spark applications is similar to that of an optimal scenario where all servers are connected to one single switch.

To complement testbed experiments, we performed large-scale simulations to study the impact of traffic stability on MegaSwitch (§5.2). For synthetic traces, MegaSwitch provides near full bisection bandwidth for stable traffic of all patterns, but does not sustain high throughput for concentrated, unstable traffic with stability period less than reconfiguration delay. However, it can improve throughput for unstable wide-spread traffic through the basemesh. For real production traces, MegaSwitch achieves 93.21% throughput of an ideal non-blocking fabric. We also find that our basemesh effectively handles highly unstable wide-spread traffic, and contributes up to 56.14% throughput improvement.

2 Background and Motivation

In this section, we first highlight the trend of high-demand wide-spread traffic in DCNs. Then we discuss why prior solutions are insufficient to support this trend.

2.1 The Trend of DCN Traffic

We use two case studies to show the traffic trend.

- **User-facing web applications:** In response to user requests, web servers push/pull contents to/from cache servers (e.g. Redis [43]). Web servers and cache servers are usually deployed in different racks, leading to intensive inter-rack traffic [42]. Cache objects are replicated in different clusters for fault-tolerance and load balancing, and web servers access these objects randomly, creating a wide-spread communication pattern overall.
- **Data-parallel processing:** Traffic of MapReduce-type applications [3, 21, 42] is shown to be heavily cluster-local (e.g. [42] reported 13.3% of the traffic is rack-local, but 80.9% cluster-local). Data is spread to many racks due to rack awareness [19], which requires

[2] Each unique wavelength corresponds to an optical port on OWS, as well as an optical transceiver on EPS.

copies of data to be stored in different racks in case of rack failures, as well as cluster-level balancing [45]. In shuffle and output phases of MapReduce jobs, wide-spread many-to-many traffic emerges among all participating servers/racks within the cluster.

Bandwidth demand is also surging as the data stored and processed in DCNs continue to grow, due to the explosion of user-generated contents such as photo/video, updates, and sensor measurements [46]. The implication is twofold: 1) Data processing applications require larger bandwidth for data exchange both within a application (e.g., in a multi-stage machine learning, workers exchange data between stages) and between applications (e.g., related applications like web search and ad recommendation share data); 2) Front-end servers need more bandwidth to fetch the ever-larger contents from cache servers to generate webpages [5]. In conclusion, optical DCN interconnects must support wide-spread, high-bandwidth demands among many racks.

2.2 Prior Proposals Are Insufficient

Prior proposals fall short in supporting the above trend of high-bandwidth, wide-spread communications among many nodes simultaneously:

- c-Through [50] and Helios [12] are seminal works that introduce wavelength division multiplexing (WDM) and optical circuit switch (OCS) into DCNs. However, they suffer from constrained rack-to-rack connectivity. At any time, high capacity optical circuits are limited to a couple of racks (e.g., 2 [50]). Reconfiguring circuits to serve communications among more racks can incur ∼10 ms circuit visit delay [12, 50].
- OSA [6] and WaveCube [7] enable arbitrary rack-to-rack connectivity via multi-hop routing over multiple circuits and EPSes. While solving the connectivity issue, they introduce severe bandwidth constraints between racks, because traffic needs to traverse multiple EPSes to destination, introducing traffic overhead and routing complexity at each transit EPS.
- Quartz [26] is an optical fiber ring that is designed as an design element to provide low latency in portions of traditional DCNs. It avoids the circuit switching delay by using fixed wavelengths, and its bisection bandwidth are also limited.
- Mordia [38] and REACToR [24] improve optical circuit switching in the temporal direction, by reducing circuit switching time from milliseconds to microseconds. This approach is effective in quickly time-sharing circuits across more nodes, but still insufficient to support wide-spread communications due to the parallel circuits available simultaneously. Furthermore, Mordia is by default a single fiber ring structure with small port density (limited by # of unique wavelengths supported on a fiber). Naïvely stacking multiple fibers via ring-selection circuits as suggested by [38] is blocking among hosts on different fibers, leading to very degraded bisection bandwidth (§5.2); whereas time-sharing multiple fibers via EPS as suggested by [11] requires complex EPS functions unavailable in existing commodity EPSes and introduces additional optical design complexities, making it hard to implement in practice (more details in §3.1).
- Free-space optics proposals [15, 18] eliminate wiring. Firefly [18] leverages free-space optics to build a wireless DCN fabric with improved cost-performance tradeoff. With low switching time and high scalability, ProjecToR [15] connects an entire DCN of tens of thousands of machines. However, they face practical challenges of real DCN environments (e.g., dust and vibration). In contrast, we implement a wired optical interconnect for thousands of machines, which is common in DCNs, and deploy real applications on it. We note that the mechanisms we developed for MegaSwitch can be employed in ProjecToR, notably the basemesh for low latency applications.

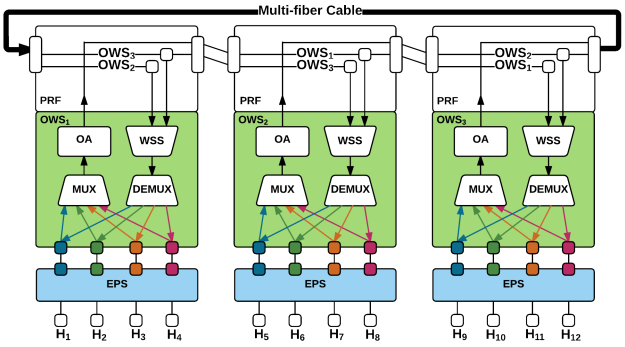


Figure 2: Example of a 3-node MegaSwitch.

3 MegaSwitch Design

In this section, we describe the design of MegaSwitch's data and control planes, as well as its fault-tolerance.

3.1 Data Plane

As in Figure 2, MegaSwitch connects multiple OWSes in a multi-fiber ring. Each fiber has only one sender. The sender broadcasts traffic on this fiber, which reaches all other nodes in one hop. Each receiver can receive traffic from all the senders simultaneously. The key of MegaSwitch is that it exploits WSS to enable non-blocking space division multiplexing among these parallel fibers.

Sending component: In OWS, for transmission, we use an optical multiplexer (MUX) to join the signals from EPS onto a fiber, and then use an optical amplifier (OA) to boost the power. Multiple wavelengths from EPS uplink ports (each with an optical transceiver at a unique wavelength) are multiplexed onto a single fiber. The multiplexed signals then travel to all the other nodes via this fiber. The OA is added before the path to compensate

the optical insertion loss caused by broadcasting, which ensures the signal strength is within the receiver sensitivity range (Detailed Power budgeting design is in §4.1). As shown in Figure 2, any signal is amplified once at its sender, and there is no additional amplification stages in the intermediate or receiver nodes. The signal from sender also does not loop back, and terminates at the last receiver on the ring (e.g., signals from OWS_1 terminates in OWS_3, so that no physical loop is formed.).

Receiving component: We use a WSS and an optical demultiplexer (DEMUX) at the receiving end. The design highlight is using WSS as a $w\times1$ wavelength multiplexer to intercept all the wavelengths from all the other nodes on the ring. In prior work [6, 11, 38], WSS was used as a $1\times w$ demultiplexer that takes 1 input fiber of k wavelengths, and outputs any subset of these k wavelengths to any of w output fibers. In MegaSwitch, WSS is repurposed as a $w\times1$ multiplexer, which takes w input fibers with k wavelengths each, and outputs a non-interfering subset of the $k\times w$ wavelengths to an output fiber. With this unconventional use of WSS, MegaSwitch ensures that any node can simultaneously choose any wavelengths from any other nodes, enabling unconstrained connections. Then, based on the demands among nodes, the WSS selects the right wavelengths and multiplexes them on the fiber to the DEMUX. In §3.2, we introduce algorithms to handle wavelength assignments. The multiplexed signals are then de-multiplexed by DEMUX to the uplink ports on EPS.

Supporting $*$-cast [51]: Unicast and multicast can be set up with a single wavelength on MegaSwitch, because any wavelength from a source can be intercepted by every node on the ring. To set up a unicast, MegaSwitch assigns a wavelength on the fiber from the source to destination, and configures the WSS at the destination to select this wavelength. Consider a unicast from H_1 to H_6 in Figure 2. We first assign wavelength λ_1 from node 1 to 2 for this connection, and configure the WSS in node 2 to select λ_1 from node 1. Then, with the routing in both EPSes configured, the unicast circuit from H_1 to H_6 is established. Further, to setup a multicast from H_1 to H_6 and H_9, based on the unicast above, we just need to additionally configure the WSS in node 3 to also select λ_1 from node 1. In addition, many-to-one (incast) or many-to-many (allcast) communications are composites of many unicasts and/or multicasts, and they are supported using multiple wavelengths.

Scalability: MegaSwitch's port count is $n\times k$.

- For n, with existing technology, it is possible to achieve 32×1 WSS (thus $n{=}w{+}1{=}33$). Furthermore, alternative optical components, such as AWGR and filter arrays can be used to achieve the same functionality as WSS+DEMUX for MegaSwitch [56]. Since 48×48 or 64×64 AWGR is available, we expect to see a $1.5\times$ or $2\times$ increase with 49 or 65 nodes on a ring, while additional splitting loss can be compensated.
- For k, C-band Dense-WDM (DWDM) link can support $k{=}96$ wavelengths at 50Ghz channel spacing, and $k{=}192$ wavelengths at 25Ghz [35]. Recent progress in optical comb source [30, 53] assures the relative wavelength accuracy among DWDM signals and is promising to enable a low power consumption DWDM transceiver array at 25Ghz channel spacing.

As a result, with existing technology, MegaSwitch supports up to $n\times k{=}33\times192{=}6336$ ports on a single ring.

MegaSwitch can also be expanded multidimensionally. In our OWS implementation (§4), two inter-OWS ports are used in the ring, and we reserve another two for future extension of MegaSwitch in another dimension. When extended bi-dimensionally (into a torus) [55], MegaSwitch scales as $n^2\times k$, supporting over 811K ports ($n{=}65$, $k{=}192$), which should accommodate modern DCNs [42, 46]. However, such large scale comes with inherent routing, management, and reliability challenges, and we leave it as future work.

MegaSwitch vs Mordia: Mordia [38] is perhaps the closest related work to MegaSwitch in terms of topology: both are constructed as a ring, and both adopt WSS. However, the key difference is: Mordia chains multiple WSSes *on* one fiber, each WSS acts as a wavelength demultiplexer to set up circuits between ports on the same fiber; whereas MegaSwitch reverses WSS as a wavelength multiplexer *across* multiple fibers to enable non-blocking connections between ports on different fibers.

We note that Farrington et al. [11] further discussed scaling Mordia with multiple fibers non-blocking-ly (referred to as Mordia-PTL below). Unlike MegaSwitch's WSS-based multi-fiber ring, they proposed to connect each EPS to multiple fiber rings in parallel, and rely on the EPS to schedule circuits for servers to support TDM across fibers. This falls beyond the capability of existing commodity switches [24]. Further, on each fiber, they allow every node to add signals to the fiber and require an optical amplifier in each node to boost the signals and compensate losses (e.g., bandpass add/drop filter loss, variable optical attenuator loss, $10/90$ splitter loss, etc.), thus they need multiple amplifiers per fiber. This degrades optical signal to noise ratio (OSNR) and makes transmission error-prone. In Figure 3, we compare OSNR between MegaSwitch and Mordia. We assume 7dBm per-channel launch power, 16dB pre-amplification gain in MegaSwitch, and 23dB boosting gain per-node in Mordia. The results show that, for MegaSwitch, OSNR maintains at $\sim$36dB and remains constant for all nodes; for Mordia, OSNR quickly degrades to 30dB after 7 hops. For validation, we have also measured the aver-

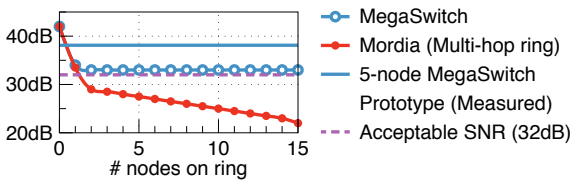


Figure 3: OSNR comparison

age OSNR on our MegaSwitch prototype (§4), which is 38.12dB after 5 hops (plotted in the figure for reference).

Furthermore, there exists a physical loop in each fiber of their design where optical signals can circle back to its origin, causing interferences if not handled properly. MegaSwitch, by design, avoids most of these problems: 1) one each fiber has only one sender and thus one stage of amplification (Power budgeting is explained in §4.1); 2) each fiber is terminated in the last node in the ring, thus there is no physical optical loop or recirculated signals (Figure 9). Therefore, MegaSwitch maintains good OSNR when scaled to more nodes. More importantly, besides above, they do not consider fault-tolerance and actual implementation in their paper [11]. In this work, we have built, with significant efforts, a functional MegaSwitch prototype with commodity off-the-shelf EPSes and our home-built OWSes.

3.2 Control Plane

As prior work [6, 12, 38, 50], MegaSwitch takes a centralized approach to configure routings in EPSes and wavelength assignments in OWSes. The MegaSwitch controller converts traffic bandwidth demands into wavelength assignments and pushes them into the OWSes. The demands can be obtained by existing traffic estimation schemes [2, 12, 50]. In what follows, we first introduce the wavelength assignment algorithms, and then design basemesh to handle unstable traffic and latency sensitive flows during reconfigurations.

3.2.1 Wavelength Assignment

In MegaSwitch, each node uses the same k wavelengths to communicate with other nodes. The control plane assigns the wavelengths to satisfy communication demands among nodes. In a feasible assignment, wavelengths from different senders must not interfere with each other at the receiver, since all the selected wavelengths to a particular receiver share the same output fiber through $w \times 1$ WSS (see Figure 2). We illustrate an assignment example in Figure 4, which has 3 nodes and each has 4 wavelengths. The demand is in (a): each entry is the number of required wavelengths of a sender-receiver pair. If the wavelengths are assigned as in (b), then two conflicts occur. A non-interfering assignment is shown in (c), where no two same wavelengths go to the same receiver.

Given the constraint, we reduce the wavelength assignment problem in MegaSwitch to an edge coloring

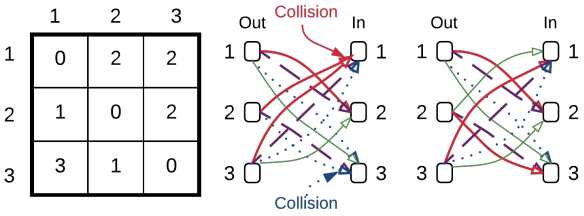


Figure 4: Wavelength assignments (n=3, k=4)**.**

problem on a bipartite multigraph. We express bandwidth demand matrix on a bipartite multigraph as Figure 4. Multiple edges between two nodes correspond to multiple wavelengths needed by them. Assume each wavelength has a unique color, then a feasible wavelength assignment is equivalent to an assignment of colors to the edges so that no two adjacent edges share the same color—exactly the edge coloring problem [9].

Edge coloring problem is $\mathcal{NP}$-complete on general graphs [6, 9], but has efficient optimal solutions on bipartite multigraphs [7, 44]. By adopting the algorithm in [7], we prove the following theorem[3] (see Appendix), which establishes the rearrangeably non-blocking property of MegaSwitch.

Theorem 1. *Any feasible bandwidth demand can be satisfied by MegaSwitch with at most k unique wavelengths.*

3.2.2 Basemesh

As WSS port count increases, MegaSwitch reconfigures at milliseconds (§4.2). To accommodate unstable traffic and latency-sensitive applications, a typical solution is to maintain a parallel electrical packet-switched fabric, as suggested by prior hybrid network proposals [12, 24, 50].

MegaSwitch emulates such hybrid network with stable circuits, providing constant connectivity among nodes and eliminating the need for an additional electrical fabric. We denote this set of wavelengths as basemesh, which should achieve two goals: 1) the number of wavelengths dedicated to basemesh, b, should be adjustable[4]; 2) With given b, guarantee low average latency for all pairs of nodes.

Essentially, basemesh is an overlay network on MegaSwitch's physical multi-fiber ring, and building such a network with the above design goals has been studied in overlay networking and distributed hash table (DHT) literature [29, 48]. Forwarding a packet on basemesh is similar to performing a look-up on a DHT network. Also, the routing table size (number of known peer addresses) of each node in DHT is analogous to the the number of wavelengths to other nodes, b. Meanwhile, our problem differs from DHT: We assume the centralized controller

[3] We note that this problem can also be reduce to the well-known row-column-row permutation routing problem [4, 39], and the reduction is also a proof of Theorem 1.

[4] The basemesh alone can be considered as a b:k (k is the number of wavelengths per fiber) over-subscribed parallel electrical switching network similar to that of [12, 24, 50].

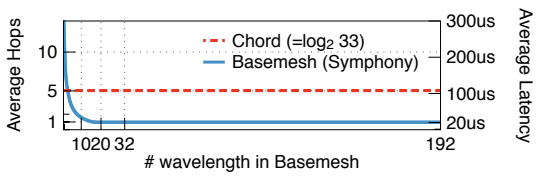


Figure 5: Average latency on basemesh.

calculates routing tables for EPSes, and the routing tables are updated for adjustments of b and peer churn (infrequent, only happens in failures §3.3).

We find the Symphony [29] topology fits our goals. For b wavelengths in the basemesh, each node first uses one wavelength to connect to the next node, forming a directed ring; then each node chooses shortcuts to $b-1$ other nodes on the ring. We adopt the randomized algorithm of Symphony [29] to choose shortcuts (drawn from a family of harmonic probability distributions, $p_n(x)=\frac{1}{xlnn}$. If a pair is selected, the corresponding entry in demand matrix increment by 1.). The routing table on the EPS of each node is configured with greedy routing that attempts to minimize the absolute distance to destination at each hop [29].

It is shown that the expected number of hops for this topology is $O(\frac{log^2(n)}{k})$ for each packet. With n=33, we plot the expected path length in Figure 5 to compare it with Chord [48] (Finger table size is 4, average path length is $O(log(n))$), and also plot the expected latency assuming $20\mu s$ per-hop latency. Notably, if $b \geq n-1$, basemesh is fully connected with 1 hop between every pair; adding more wavelength cannot reduce the average hop count, but serve to reduce congestion.

As we will show in §5.2, while basemesh reduces the worst-case bisection bandwidth, the average bisection bandwidth is unaffected, and it brings two benefits:

- It increases throughput for rapid-changing, unpredictable traffic when the traffic stability period is less than the control plane delay, as shown in §5.2.
- It mitigates the impact of inaccurate demand estimation by providing consistent connection. Without basemesh, when the demand between two nodes is mistakenly estimated as 0, they are effectively cut-off, which is costly to recover (flows need to wait for wavelength setup).

Thanks to the flexibility of MegaSwitch, b is adjustable to accommodate varying traffic instability (§5.2), or be partially/fully disabled for higher traffic imbalance if applications desire more bandwidth in part of network [42]. In this paper, we present b as a knob for DCN operators, and intend to explore the problem of optimal configuration of b as future work.

Basemesh vs Dedicated Topology in ProjecToR [15]: Basemesh is similar to the dedicated topology in ProjecToR, which is a set of static connections for low latency traffic. However, its construction is based on the optimization of weighted path length using traffic distribution from past measurements, which cannot provide average latency guarantee for an arbitrary pair of racks. In comparison, basemesh can provide consistent connections between all pairs with proven average latency. It is possible to configure basemsesh in ProjecToR as its dedicated topology, which better handles unpredictable, latency-critical traffic.

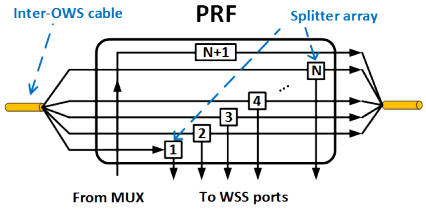


Figure 6: PRF module layout.

3.3 Fault-tolerance

We consider the following failures in MegaSwitch.

OWS failure: We implement the OWS box so that the node failure will not affect the operation of other nodes on the ring. As shown in Figure 6, the integrated passive component (Passive Routing Fabric, PRF) handles the signal propagation across adjacent nodes, while the passive splitters copy the traffic to local active switching modules (i.e., WSS and DEMUX). When one OWS loses its function (e.g., power loss) in its active part, the traffic traveling across the failure node from/to other nodes is not affected, because the PRF operates passively and still forwards the signals. Thus the failure is isolated within the local node. We design the PRF as a modular block that can be attached and detached from active switching modules easily. So one can recover the failed OWS without the service interruption of the rest network by simply swapping its active switching module.

Port failure: When the transceiver fails in a node, it causes the loss of capacity only at the local node. We have implemented built-in power detectors in OWS to notify the controller of such events.

Controller failure: In control plane, two or more instances of the controller are running simultaneously, with one leader. The fail-over mechanism follows VRRP [20].

Cable failure: Cable cut is the most difficult failure, and we design[5] redundancy and recovery mechanisms to handle one cable cut. The procedure is analogous to ring protection in SONET [49] by 2 sets of fibers. We propose directional redundancy in MegaSwitch, as shown in Figure 7, the fiber can carry the traffic from/to both east (primary) and west (secondary) sides of the OWS by select-

[5]This design has not been implemented in the current prototype, thus the power budgeting on the prototype (§4.1) does not account for the fault-tolerance components on the ring. The components for this design are all commercially available, and can be readily incorporated into future iterations of MegaSwitch.

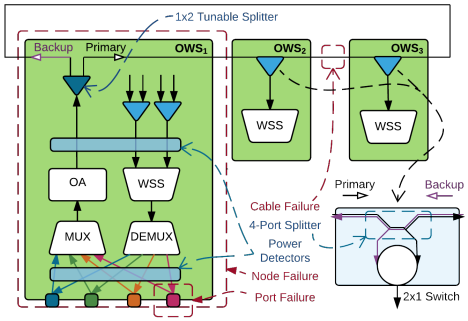


Figure 7: MegaSwitch fault-tolerance.

ing a direction at the sending 1×2 optical tunable splitter[6], and receiving 2×1 optical switches. The splitters and switches are in the PRF module. When there is no cable failure, sending and receiving switches both select the primary direction, thus preventing optical looping. If one cable cut is detected by the power detectors, the controller is notified, and it instructs the OWSes on the ring to do the following: 1) The sending tunable splitter in every OWS transmits on its fiber on both directions[7], so that the signal can reach all the OWSes; 2) The receiving switch for each fiber at the input of WSS selects the direction where there is still incoming signal. Then the connectivity can be restored. A passive 4-port fix splitter is added before the receiving switch as a part of the ring, and it guides the signal from primary and backup directions to different input ports of the 2×1 switch. For more than one cut, the ring is severed into two or more segments, and connectivity between segments cannot be recovered until cable replacement.

4 Implementation

4.1 Implementation of OWS

To implement MegaSwitch and facilitate real deployment, we package all optical devices into an 1RU (Rack Unit) switch box, OWS (Figure 2). The OWS is composed of the sending components (MUX+OA) and the receiving components (WSS+DEMUX), as described in §3. We use a pair of array waveguide grating (AWG) as MUX and DEMUX for DWDM signals.

To simplify the wiring, we compact all the 1×2 drop-continue splitters into PRF module, as shown in Figure 6. This module can be easily attached to or detached from active switching components in the OWS. To compensate component losses along the ring, we use a single stage 12dB OA to boost the DWDM signal before transmitting the signals, and the splitters in PRF have different splitting ratios. Using the same numbering in Figure 6, the splitting ratio of i-th splitter is 1:$(i-1)$ for $i>1$. As an example, the power splitting on one fiber from node 1 is shown in Figure 9. At each receiving transceiver,

[6]It is implementable with a common Mach-Zehnder interferometer.

[7]Power splitting ratio must also be re-configured to keep the signal within receiver sensitivity range for all receiving OWSes

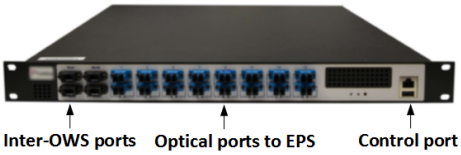


Figure 8: The OWS box we implemented.

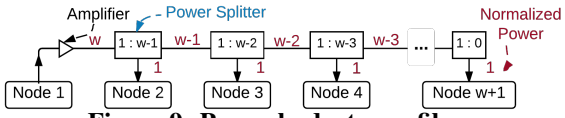


Figure 9: Power budget on a fiber

the signal strength is ∼−9dBm per channel. The PRF configuration is therefore determined by w (WSS radix).

OWS receives DWDM signals from all other nodes on the ring via its WSS. We adopt 8×1 WSS from CoAdna Photonics in the prototype, which is able to select wavelength signals from at most 8 nodes. Currently we use 4 out of 8 WSS ports for a 5-node ring.

Our OWS box provides 16 optical ports, and 8 are used in the MegaSwitch prototype. Each port maps to a particular wavelength from 190.5THz to 193.5THz at 200GHz channel spacing. InnoLight 10GBASE-ER DWDM SFP+ transceivers are used to connect EPSes to optical ports on OWSes. With the total broadcasting loss at 10.5dB for OWS, 4dB for WSS, 2.5dB for DEMUX, and 1dB for cable connectors, the receiving power is −9dBm ∼−12dBm, well within the sensitivity range of our transceivers. The OWS box we implemented (Figure 8) has 4 inter-OWS ports, 16 optical ports, and 1 Ethernet port for control. For the 4 inter-OWS ports, two are used to connect other OWSes to construct the MegaSwitch ring, and the other two are reserved for redundancy and further scaling (§3.1).

Each OWS is controlled by a Raspberry Pi [37] with 700MHz ARM CPU and 256MB memory. OWS receives the wavelength assignments (in UDP packets) via its Ethernet management interface connected to a separate, electrical control plane network, and configures WSS via GPIO pins.

4.2 MegaSwitch Prototype

We constructed a 5-node MegaSwitch with 5 OWS boxes, as is shown in Figure 10. The OWSes are simply connected to each other through their inter-OWS ports using ring cables containing multiple fibers. The two EPSes we used are Broadcom Pronto-3922 with 48×10 Gbps Ethernet (GbE) ports. For one EPS, we fit 24 ports with 3 sets of transceivers of 8 unique wavelengths to connect to 3 OWSes, and the rest 24 ports are connected to the servers. For the other, we only use 32 ports with 16 ports to OWSes and 16 to servers. We fill the capacity with 40×10GbE interfaces on 20 Dell PowerEdge R320 servers (Debian 7.0 with Kernel 3.18.19), each with a Broadcom NetXtreme II 10GbE network interface card.

The control plane network connected by a Broadcom Pronto-3295 Ethernet switch. The Ethernet management

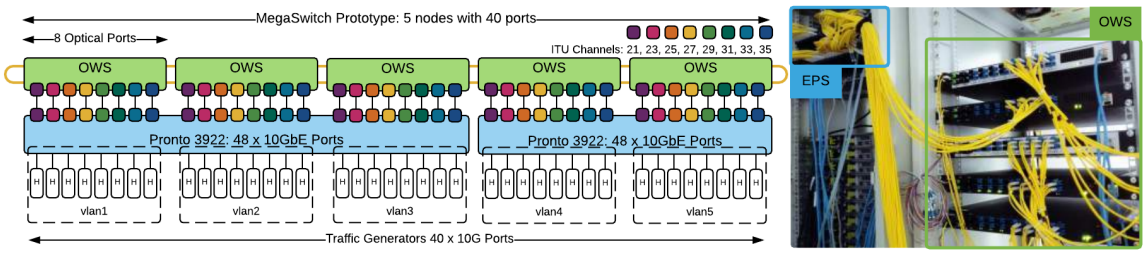


Figure 10: The MegaSwitch prototype implemented with 5 OWSes.

ports of EPS and OWS are connected to this switch. The MegaSwitch controller is hosted in a server also connected to the control plane switch. The EPSes work in Open vSwitch [36] mode, and are managed by Ryu controller [34] hosted in the same server as the MegaSwitch controller. To emulate 5 OWS-EPS nodes, we divided the 40 server-facing EPS ports into 5 VLANs, virtually representing 5 racks. The OWS-facing ports are given static IP addresses, and we install Openflow [31] rules in the EPS switches to route packets between VLANs in Layer 3. In the experiments, we refer to an OWS and a set of 8 server ports in the same VLAN as a node.

Reconfiguration speed: MegaSwitch's reconfiguration hinges on WSS, and the WSS switching speed is supposedly microseconds with technology in [38]. However, in our implementation, we find that as WSS port count increases (required to support more wide-spread communication), e.g., w=8 in our case, we can no longer maintain 11.5μs switching seen by [38]. This is mainly because the port count of WSS made with digital light processing (DLP) technology used in [38] is currently not scalable due to large insertion loss. As a result, we choose the WSS implemented by an alternative Liquid Crystal (LC) technology, and the observed WSS switching time is $\sim 3ms$. We acknowledge that this is a hard limitation we need to confront in order to scale. We identify that there exist several ways to reduce this time [14, 23].

We note that, with current speed on our prototype, MegaSwitch is still possible to satisfy the need of some production DCNs, as a recent study [42] of DCN traffic in Facebook suggests, with effective load balancing, traffic demands are stable over sub-second intervals. On the other hand, even if μs switching is achieved in later versions of MegaSwitch, it is still insufficient for high-speed DCNs (40/100G or beyond). Following §3.2.2, in our prototype, we address this problem by activating basemesh, which mimics hybrid network without resorting to an additional electrical network.

5 Evaluation

We evaluate MegaSwitch with testbed experiments (with synthetic patterns(§5.1.1)) and real applications(§5.1.2)), as well as large-scale simulations (with synthetic patterns and productions traces(§5.2)). Our main goals are to: 1) measure the basic metrics on MegaSwitch's data plane and control plane; 2) understand the performance of real applications on MegaSwitch; 3) study the impact of control plane latency and traffic stability on throughput, and 4) assess the effectiveness of basemesh.

Summary of results is as follows:

- MegaSwitch supports wide-spread communications with full bisection bandwidth among all ports when wavelength are configured. MegaSwitch sees ~20ms reconfiguration delay with ~3ms for WSS switching.
- We deploy real applications, Spark and Redis, on the prototype. We show that MegaSwitch performs similarly to the optimal scenario (all servers under a single EPS) for data-intensive applications on Spark, and maintains uniform latency for cross-rack queries for Redis due to the basemesh.
- Under synthetic traces, MegaSwitch provides near full bisection bandwidth for stable traffic (stability period $\geq$100ms) of all patterns, but cannot achieve high throughput for concentrated, unstable traffic with stability period less than reconfiguration delay. However, increasing the basemesh capacity effectively improves throughput for highly unstable wide-spread traffic.
- With realistic production traces, MegaSwitch achieves $>90\%$ throughput of an ideal non-blocking fabric, despite its 20ms total wavelength reconfiguration delay.

5.1 Testbed Experiments

5.1.1 Basic Measurements

Bisection bandwidth: We measure the bisection throughput of MegaSwitch and its degradation during wavelength switching. We use the $40\times$10GbE interfaces on the prototype to form dynamic all-to-all communication patterns (each interface is referred as a host). Since traffic from real applications may be CPU or disk I/O bound, to stress the network, we run the following synthetic traffic used in Helios [12]:

- **Node-level Stride (NStride):** Numbering the nodes (EPS) from 0 to $n-1$. For the i-th node, its j-th host initiates a TCP flow to the j-th host in the $(i+l \bmod n)$-th node, where l rotates from 1 to n every t seconds (t is the traffic stability period). This pattern tests the response to abrupt demand changes between nodes, as the traffic from one node completely shifts to another node in a new period.

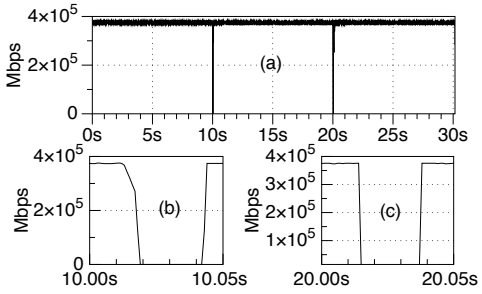


Figure 11: Experiment: Node-level stride

- **Host-level Stride (HStride):** Numbering the hosts from 0 to $n\times k-1$, the i-th host sends a TCP flow to the $(i+k+l \bmod (n\times k))$-th host, where l rotates from 1 to $\lceil k/2 \rceil$ every t seconds. This pattern showcases the gradual demand shift between nodes.
- **Random:** A random perfect matching between all the hosts is generated every period. Every host sends to its matched host for t seconds. The pattern showcases the wide-spread communication, as every node communicates with many nodes in each period.

For this experiment, the basemesh is disabled. We set traffic stability t=10s, and the wavelength assignments are calculated and delivered to the OWS when demand between 2 nodes changes. We study the impact of stability t later in §5.2.

As shown in Figure 11 (a), MegaSwitch maintains full bandwidth of 40×10Gbps when traffic is stable. During reconfigurations, we see the corresponding throughput drops: NStride drops to 0 since all the traffic shifts to a new node; HStride's performance is similar to Figure 11 (a), but drops by only 50Gbps because one wavelength is reconfigured per rack. The throughput resumes quickly after reconfigurations.

We further observe a $\sim 20ms$ gap caused by the wavelength reconfiguration at 10s and 20s in the magnified (b) and (c) of Figure 11. Transceiver initialization delay contributes ~10ms[8], and the remaining can be broken down to EPS configuration ($\sim 5ms$), WSS switching ($\sim 3ms$), and control plane delays ($\sim 4ms$). (Please refer to Appendix for detailed measurement methods and results.)

5.1.2 Real Applications on MegaSwitch

We now evaluate the performance of real applications on MegaSwitch. We use Spark [54], a data-parallel processing application, and Redis [43], a latency-sensitive in-memory key-value store. For this experiment, we form the basemesh with 4 wavelengths, and the others are allocated dynamically.

Spark: We deploy Spark 1.4.1 with Oracle JDK 1.7.0_25 and run three jobs: WikipediaPageRank[9], K-Means[10], and WordCount[11]. We first connect all the 20 servers to a single ToR EPS and run the applications, which establishes the optimal network scenario because all servers are connected with full bandwidth. We record the time series (with millisecond granularity) of bandwidth usage of each server when it is running, and then convert it into two series of averaged bandwidth demand matrices of $0.1s$ and 1s intervals respectively. Then, we connect the servers back to MegaSwitch, and re-run the applications using these two series as inputs to update MegaSwitch every 0.1s and 1s intervals accordingly.

[8] Our transceiver's receiver Loss of Signal Deassert is -22dBm.

[9] A PageRank instance using a 26GB dataset [32]

[10] A clustering algorithm that partitions a dataset into K clusters. The input is Wikipedia Page Traffic Statistics [47].

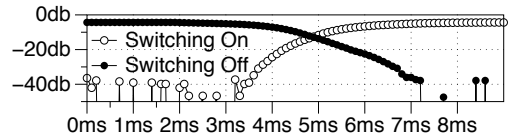


Figure 12: WSS switching speed.

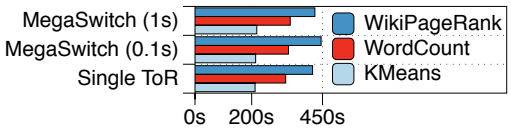


Figure 13: Completion time for Spark jobs

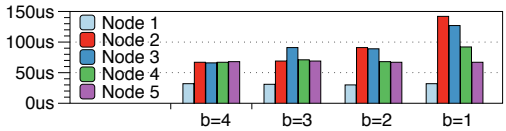


Figure 14: Redis intra/cross-rack query completion

We plot the job completion times in Figure 13. With ~20ms reconfiguration delay and 0.1s interval, the bandwidth efficiency is expected to be $(0.1-0.02)/0.1=80\%$ if every wavelength changes for each interval. However, MegaSwitch performs almost the same as if the servers are connected to a single EPS (on average 2.21% worse). This is because, as we observed, the traffic demands of these Spark jobs are stable: e.g., for K-Means, the number of wavelength reassignments is only 3 and 2 times for the update periods of 0.1 and 1s, respectively. Most wavelengths do not need reconfiguration, and thus provide uninterrupted bandwidth during the experiments. The performance of both update periods is similar, but the smaller update interval has slightly worse performance due to one more wavelength reconfiguration. For example, in WikiPageRank, MegaSwitch with 1s update interval shows 4.94% less completion time than that with 0.1s. We further examine the relationship between traffic stability and MegaSwitch's control latencies in §5.2.

Redis: For this Redis in-memory key-value store experiment, we initiate queries to a Redis server in the first node from the servers in all 5 nodes, with a total number of 10^6 SET and GET requests. Key space is set to 10^5. Since the basemesh provides connectivity between all the nodes, Redis is not affected by any reconfigura-

[11] It counts words in a 40G dataset with 7.153×10^9 words.

tion latency. The average query completion times from servers in different racks are shown in Figure 14. Query latency depends on hop count. With 4 wavelengths in basemesh (b=4), Redis experiences uniform low latencies for cross-rack queries, because they only traverse 2 EPSes (hops). For b=3, queries also traverse 2 hops, except the ones from node 3. When b=1, basemesh is a ring, and the worst case hop count for a query is 5. Therefore, for latency-critical bandwidth-insensitive applications like Redis, setting b=n−1 guarantees uniform latency between all nodes on MegaSwitch. In comparison, for other related architectures, the number of EPS hops for cross-rack queries can reach 3 (Electrical/Optical hybrid designs [12, 24, 50]) or 5 (Pure electrical designs [1, 16, 25]) for cross-rack queries.

5.2 Large Scale Simulations

Existing packet-level simulators, such as ns-2, are time consuming to run at 1000+-host scale [2], and we are more interested in traffic throughput rather than per-packet behavior. Therefore, we implemented a flow-level simulator to perform simulations at larger scales. Flows on the same wavelength share the bandwidth in a max-min fair manner. The simulation runs in discrete time ticks with the granularity of millisecond. We assume a proactive controller: it is informed of the demand change in the next traffic stability period and runs the wavelength assignment algorithm before the change, therefore it configures the wavelengths every t seconds with a total reconfiguration delay of 20ms (§5.1.1). Unless specified otherwise, basemesh is configured with b=32.

We use synthetic patterns in §5.1.1 as well as realistic traces from production DCNs in our simulations. For the synthetic patterns, we simulate a 6336-port MegaSwitch (n=33, k=192), and each pattern runs for 1000 traffic stability periods (t). For the realistic traces, we simulate a 3072-ports MegaSwitch (n=32, k=96) to match the scale of the real cluster. We replay the realistic traces as dynamic job arrivals and departures.

- **Facebook:** Hive/MapReduce trace is collected by Chowdhury et al. [8] from a 3000-server, 150-rack Facebook cluster. For each shuffle, the trace records: start time, senders, receivers, and the received bytes. The trace contains more than 500 shuffles (7×10^5 flows). Shuffle sizes vary from 1MB to 10TB, and the number of flows per shuffle varies from 1 to 2×10^4.
- **IDC:** We collected 2-hour running trace from the Hadoop cluster (3000-server) of a large Internet service company. The trace records shuffle tasks (the same information as above is collected), as well as HDFS read tasks (sender, receiver, and size are collected). We refer to them as **IDC-Shuffle** and **IDC-HDFS** when used separately. The trace contains more than 1000 shuffle and read tasks, respectively (1.6×10^6 flows). The size of shuffle and read varies from 1MB to 1TB, and the number of flows per shuffle is between 1 and 1.7×10^4.

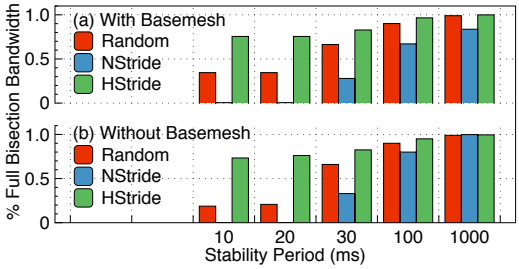


Figure 15: Throughput vs traffic stability

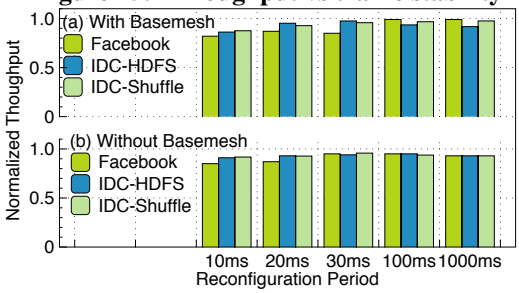


Figure 16: Throughput vs reconfiguration frequency

Impact of traffic stability: We vary the traffic stability period t from 10ms to 1s for the synthetic patterns, and plot the average throughput with and without basemesh in Figure 15. We make three observations: 1) If stability period is ≤20ms (reconfiguration delay), traffic patterns that need more reconfigurations have lower throughput, and NStride suffers the worse throughput; 2) All the three patterns see the throughput steadily increasing to full bisection bandwidth with longer stability period; 3) Although basemesh reserves wavelengths to maintain connectivity, throughput of different patterns is not negatively affected, except for NStride, which cannot achieve full bisection bandwidth even for stable traffic (t=1s), since the basemesh takes 32 wavelengths.

We then analyze the performance of different patterns in detail. NStride has near zero throughput when t=10ms and 20ms, as all the wavelengths must break down and reconfigure for each period. Basemesh does not help much for NStride: when the wavelengths are not yet available, basemesh can serve only $1/192$ of the demand. HStride reaches ∼75% full bisection bandwidth when the stability period is 10ms, because on average $3/4$ of its demands stay the same between consecutive periods, and demands in the new period reuse some of the previous wavelengths. Random pattern is wide-spread and requires more reconfigurations in each period than HStride. Thus it also suffers from unstable traffic, with 18.5% full-bisection bandwidth for t=10ms without basemesh. However, it benefits from basemesh the most, achieving 34.1% full-bisection bandwidth for t=10ms, because the flows between two nodes need not to wait for reconfigurations.

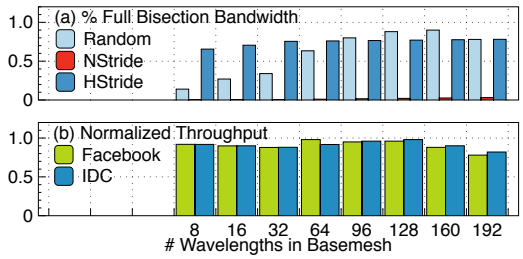


Figure 17: Handling unstable traffic with basemesh

In summary, MegaSwitch provides near full-bisection bandwidth for stable traffic of all patterns, but cannot achieve high throughput for concentrated, unstable traffic (NStride with stability period smaller than reconfiguration delay. We note that such pattern is designed to test the worst-case behavior of MegaSwitch, and is unlikely to occur in production DCNs.).

Impact of reconfiguration frequency: We vary the reconfiguration frequency to study its impact on throughput using realistic traces. In Figure 16, we collect the throughput of the replayed traffic traces[12], and then normalize them to the throughput of the same traces replayed on a non-blocking fabric with the same port count. The normalized throughput shows how MegaSwitch approaches non-blocking.

From the results, we find that MegaSwitch achieves 93.21% and 90.84% normalized throughput on average with and without basemesh respectively. This indicates that our traces collected from the production DCNs are very stable, thus can take advantage of the high bandwidth of optical circuits. This aligns well with traffic statistics in another study [42], which suggests, with good load balancing, the traffic is stable on sub-second scale, thus is suitable for MegaSwitch. We also observe that the traffic is wide-spread for realistic traces: for every period, a rack in Facebook, IDC-HDFS & IDC-Shuffle, talks with 12.1, 4.5 & 14.6 racks on average, respectively. Limited by rack-to-rack connectivity, other optical structures are less effective for such patterns.

Impact of adjusting basemesh capacity: For rapidly changing traffic, MegaSwitch can improve its throughput with more wavelengths to basemesh. In Figure 17, we measure the throughput of synthetic patterns (stability period is 10ms) and production traces. The production traces feature dynamic arrival/departure of tasks.

For NStride and HStride (Figure 17 (a)), since the traffic of each node is destined to one or two other nodes, increasing basemesh does not benefit them much. In contrast, for Random, each node sends to many nodes (i.e., wide-spread), increasing the capacity of basemesh, b, from 8 wavelengths to 32 wavelengths increases the throughput by 20.6%; Further increasing b to 160 can increase the throughput by 56.1%. But this trend is not monotonic: if all the wavelengths are in basemesh (b=192), and the demands between nodes larger than 6 wavelengths cannot be supported. This is also observed for the realistic traces in Figure 17 (b): when >4 wavelengths are allocated to basemesh, the normalized throughput decreases.

[12]The wavelength schedules are obtain in the same way as Spark experiments in §5.1.2.

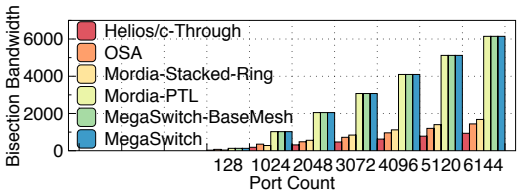


Figure 18: Average bisection throughput.

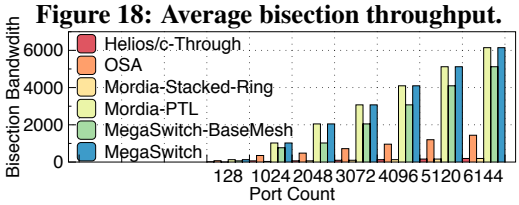


Figure 19: Worst-case bisection throughput.

In summary, for rapidly changing traffic, basemesh is an effective and adjustable tool for MegaSwitch to handle wide-spread demand fluctuation. As a comparison, hybrid structures [12, 24, 50] cannot dynamically adjust the capacity of the parallel electrical fabrics.

Bisection bandwidth: To compare MegaSwitch with other optical proposals, we calculate the average and worst-case bisection bandwidths of MegaSwitch and related proposals in Figure 18 & 19, respectively. The unit of y-axis is bandwidth per wavelength. We generate random permutation of port pairing for 10^4 times to compute the average bisection bandwidth for each structure, and design the worst case scenario for them: for Mordia, OSA, and Helios/c-Through, the worst-case scenario is when every node talks to all other nodes[13], because a node can only directly talk to a limited number of other nodes in these structures. We calculate the total throughput of simultaneous connections between ports. We rigorously follow respective papers to scale port counts from 128 to 6144. For example, OSA at 1024 port uses a 16-port OCS for 16 racks each with 64 hosts; Mordia at 6144 ports uses a 32-port OCS to connect 32 rings each with 192 hosts.

We observe: 1) MegaSwitch and Mordia-PTL can achieve full bisection bandwidth at high port count in both average and the worst case, while other designs, including Mordia-Stacked-Ring, cannot. Both structures are expected to maintain full bisection bandwidth with future scaling in optical devices with larger WSS radix and wavelengths per-fiber; 2) Basemesh reduces the worst-case bisection bandwidth by 16.7% for b=32, but full bisection bandwidth is still achieved on average.

[13]A node refers to a ring in Mordia, a rack in OSA/Helios/c-Through

6 Cost of MegaSwitch

Complexity analysis: The absolute costs of optical proposals are difficult to calculate, as it depends on multiple factors, such as market availability, manufacturing costs, etc. For example, our PRF module can be printed as a planar lightwave circuit, and mass-production can push its cost to that of common printed circuit boards [10]. Thus, instead of directly calculating the costs based on price assumptions, we take a comparative approach and analyze the structural complexity of MegaSwitch and closely related proposals.

In Table 1, we compare the complexity of different optical structures at the same port count (3072). For OSA, it translates to 32 racks, 96 DWDM wavelengths, and a 128-port OCS (Optical Circuit Switch). ToR degree is set to 4 [6]. Quartz's port count is limited to the wavelength per fiber (k) [26]: so a Quartz element with k=96 is essentially a 96-port switch, and we scale it to 3072 ports by composing 160 Quartz elements[14] in a FatTree [1] topology (with 32 pods and 32 cores). For Mordia, we use its microsecond 1×4 WSS and stack 32 rings via a 32 port OCS to reach 3072 ports. Mordia-PTL [11] is configured in the same way as Mordia, with the only difference that each EPS is directly connected to 32 rings in parallel without using OCS. Both Mordia and Mordia-PTL have $96/4$=24 stations on each ring. MegaSwitch is configured with 32 nodes and 96 wavelengths per node. Finally, we list a FatTree constructed with 24-port EPS, with totally $24^3/4$=3456 ports and 720 EPSes. We assume that, within the FatTree fabric, all the EPSes are connected via transceivers.

From the table, we find that compared to other optical structures, MegaSwitch supports the same port count with less optical components. Compared to all the optical solutions, FatTree uses 2× optical transceivers (non-DWDM) at the same scale.

Transceiver sost: Our prototype uses commercial 10Gb-ER DWDM SFP+ transceivers, which are ∼10× more expensive per bit per second than the non-DWDM modules using PSM4. This is because ER optics is usually used for long-range optical networks, rather than short-range networks within a DCN. Our choice of using ER optics is solely due to its ability to use DWDM wavelengths, not its power output or 40KM reach. If MegaSwitch or similar optical interconnects are widely adopted in future DCNs, we expect that DWDM transceivers customized for DCNs will be used instead of current PSM4 modules. Such transceivers are being developed [22], and due to relaxed requirements of short-range DCN, the cost is expectedly lower. Even if the customized DWDM transceivers are still more expensive than non-DWDM ones, since FatTree needs many more EPSes (and thus transceivers), and the cost difference will grow as the network scales [38].

Components	OSA	Quartz	Mordia	Mordia-PTL	MegaSwitch	FatTree
Amplifier	0	6144	768	768	32	0
WSS	32 (1×4)	960	768 (1×4)	768 (1× 4)	32 (32×1)	0
WDM Filter	0	0	768	768	0	0
OCS	1×128-port	0	1×32-port	0	0	0
Circulator	3072	0	0	0	0	0
Transceivers	3072	3072	3072	3072	3072	6912

Table 1: Comparison of optical complexity at the same scale (3072 ports) in optical components used.

Amplifier cost: With only one stage of amplification on each fiber, MegaSwitch can maintain good OSNR at large scale (Figure 3). Therefore, MegaSwitch needs much fewer OAs (optical amplifier) at the same scale. The tradeoff is that they must be more powerful, as the total power to reach the same number of ports cannot be reduced significantly. Although unit cost of a powerful OA is higher[15], we expect the total cost to be similar or lower, as MegaSwitch requires much fewer OAs (24× fewer than Mordia and 192× fewer than Quartz).

WSS cost: In Table 1, MegaSwitch uses 32×1 WSS, and one may wonder its cost versus 1×4 WSS. In fact, our design allows for low cost WSS, as transceiver link budget design does not need to consider optical hopping, thus the key specifications can be relaxed (e.g., bandwidth, insertion loss, and polarization dependent loss requirements). Liquid Crystal (LC) technology is used in our WSS as it is easier to cost-effectively scale to more ports. As the majority of the components in a 32×1 WSS are same as a 1×4 WSS, the per-port cost of 32×1 WSS is about 4 times lower than that of a current 1×4 WSS [6, 38]. In the future, silicon photonics (e.g., matrix switch by ring resonators [13]) can improve the integration level [52] and further reduce the cost.

7 Conclusion

We presented MegaSwitch, an optical interconnect that delivers rearrangeably non-blocking communication to 30+ racks and 6000+ servers. We have implemented a working 5-rack 40-server prototype, and with experiments on this prototype as well as large-scale simulations, we demonstrated the potential of MegaSwitch in supporting wide-spread, high-bandwidth demands workloads among many racks in production DCNs.

Acknowledgements: This work is supported in part by Hong Kong RGC ECS-26200014, GRF-16203715, GRF-613113, CRF-C703615G, China 973 Program No.2014CB340303, NSF CNS-1453662, CNS-1423505, CNS-1314721, and NSFC-61432009. We thank the anonymous NSDI reviewers and our shepherd Ratul Mahajan for their constructive feedback and suggestions.

[14] We assume a Quartz ring of 6 wavelength add/drop multiplexers [26] in each element, thus 6×160=960 in total.

[15] For example, powerful OA can be constructed by a series of less powerful ones.

References

[1] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A scalable, commodity data center network architecture. In *ACM SIGCOMM* (2008).

[2] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. Hedera: Dynamic flow scheduling for data center networks. In *USENIX NSDI* (2010).

[3] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center tcp (dctcp). *ACM SIGCOMM Computer Communication Review 41*, 4 (2011), 63–74.

[4] ANNEXSTEIN, F., AND BAUMSLAG, M. A unified approach to off-line permutation routing on parallel networks. In *ACM SPAA* (1990).

[5] CEGLOWSKI, M. The website obesity crisis. "http://idlewords.com/talks/website_obesity.htm", 2015.

[6] CHEN, K., SINGLA, A., SINGH, A., RAMACHANDRAN, K., XU, L., ZHANG, Y., WEN, X., AND CHEN, Y. Osa: An optical switching architecture for data center networks with unprecedented flexibility. In *USENIX NSDI* (2012).

[7] CHEN, K., WEN, X., MA, X., CHEN, Y., XIA, Y., HU, C., AND DONG, Q. Wavecube: A scalable, fault-tolerant, high-performance optical data center architecture. In *IEEE INFOCOM* (2015).

[8] CHOWDHURY, M., ZHONG, Y., AND STOICA, I. Efficient coflow scheduling with varys. In *ACM SIGCOMM* (2014).

[9] COLE, R., AND HOPCROFT, J. On edge coloring bipartite graphs. *SIAM Journal on Computing 11*, 3 (1982), 540–546.

[10] DOERR, C. R., AND OKAMOTO, K. Advances in silica planar lightwave circuits. *Journal of lightwave technology 24*, 12 (2006), 4763–4789.

[11] FARRINGTON, N., FORENCICH, A., PORTER, G., SUN, P.-C., FORD, J., FAINMAN, Y., PAPEN, G., AND VAHDAT, A. A multiport microsecond optical circuit switch for data center networking. In *IEEE Photonics Technology Letters* (2013).

[12] FARRINGTON, N., PORTER, G., RADHAKRISHNAN, S., BAZZAZ, H. H., SUBRAMANYA, V., FAINMAN, Y., PAPEN, G., AND VAHDAT, A. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *ACM SIGCOMM* (2010).

[13] GAETA, A. L. All-optical switching in silicon ring resonators. In *Photonics in Switching* (2014).

[14] GEIS, M., LYSZCZARZ, T., OSGOOD, R., AND KIMBALL, B. 30 to 50 ns liquid-crystal optical switches. In *Optics express* (2010).

[15] GHOBADI, M., MAHAJAN, R., PHANISHAYEE, A., DEVANUR, N., KULKARNI, J., RANADE, G., BLANCHE, P.-A., RASTEGARFAR, H., GLICK, M., AND KILPER, D. Projector: Agile reconfigurable data center interconnect. In *ACM SIGCOMM* (2016).

[16] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VL2: A scalable and flexible data center network. In *ACM SIGCOMM* (2009).

[17] HALPERIN, D., KANDULA, S., PADHYE, J., BAHL, P., AND WETHERALL, D. Augmenting data center networks with multi-gigabit wireless links. In *SIGCOMM* (2011).

[18] HAMEDAZIMI, N., QAZI, Z., GUPTA, H., SEKAR, V., DAS, S. R., LONGTIN, J. P., SHAH, H., AND TANWERY, A. Firefly: A reconfigurable wireless data center fabric using free-space optics. In *ACM SIGCOMM* (2014).

[19] HEDLUND, B. Understanding hadoop clusters and the network. "http://bradhedlund.com/", 2011.

[20] HINDEN, R. Virtual router redundancy protocol (vrrp). *RFC 5798* (2004).

[21] KANDULA, S., SENGUPTA, S., GREENBERG, A., PATEL, P., AND CHAIKEN, R. The nature of datacenter traffic: Measurements and analysis. In *ACM IMC* (2009).

[22] LIGHTWAVE. Ranovus unveils optical engine, 200-gbps pam4 optical transceiver. "https://goo.gl/QbRUd4", 2016.

[23] LIN, X.-W., HU, W., HU, X.-K., LIANG, X., CHEN, Y., CUI, H.-Q., ZHU, G., LI, J.-N., CHIGRINOV, V., AND LU, Y.-Q. Fast response dual-frequency liquid crystal switch with photo-patterned alignments. In *Optics letters* (2012).

[24] LIU, H., LU, F., FORENCICH, A., KAPOOR, R., TEWARI, M., VOELKER, G. M., PAPEN, G., SNOEREN, A. C., AND PORTER, G. Circuit switching under the radar with reactor. In *USENIX NSDI* (2014).

[25] LIU, V., HALPERIN, D., KRISHNAMURTHY, A., AND ANDERSON, T. F10: A fault-tolerant engineered network. In *NSDI* (2013).

[26] LIU, Y. J., GAO, P. X., WONG, B., AND KESHAV, S. Quartz: a new design element for low-latency dcns. In *ACM SIGCOMM* (2014).

[27] LOVÁSZ, L., AND PLUMMER, M. D. *Matching theory*, vol. 367. American Mathematical Soc., 2009.

[28] MALKHI, D., NAOR, M., AND RATAJCZAK, D. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing* (2002), ACM, pp. 183–192.

[29] MANKU, G. S., BAWA, M., RAGHAVAN, P., ET AL. Symphony: Distributed hashing in a small world. In *USENIX USITS* (2003).

[30] MARTINEZ, A., CALÒ, C., ROSALES, R., WATTS, R., MERGHEM, K., ACCARD, A., LELARGE, F., BARRY, L., AND RAMDANE, A. Quantum dot mode locked lasers for coherent frequency comb generation. In *SPIE OPTO* (2013).

[31] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. Openflow: Enabling innovation in campus networks. *ACM Computer Communication Review* (2008).

[32] METAWEB-TECHNOLOGIES. Freebase wikipedia extraction (wex). http://download.freebase.com/wex/, 2010.

[33] MISRA, J., AND GRIES, D. A constructive proof of vizing's theorem. *Information Processing Letters 41*, 3 (1992), 131–133.

[34] OSRG. Ryu sdn framework. "https://osrg.github.io/ryu/", 2017.

[35] PASQUALE, F. D., MELI, F., GRISERI, E., SGUAZZOTTI, A., TOSETTI, C., AND FORGHIERI, F. All-raman transmission of 192 25-ghz spaced wdm channels at 10.66 gb/s over 30 x 22 db of tw-rs fiber. In *IEEE Photonics Technology Letters* (2003).

[36] PFAFF, B., PETTIT, J., AMIDON, K., CASADO, M., KOPONEN, T., AND SHENKER, S. Extending networking into the virtualization layer. In *ACM Hotnets* (2009).

[37] PI, R. An arm gnu/linux box for $ 25. *Take a byte* (2012).

[38] PORTER, G., STRONG, R., FARRINGTON, N., FORENCICH, A., SUN, P.-C., ROSING, T., FAINMAN, Y., PAPEN, G., AND VAHDAT, A. Integrating microsecond circuit switching into the data center. In *ACM SIGCOMM* (2013).

[39] QIAO, C., AND MEI, Y. Off-line permutation embedding and scheduling in multiplexed optical networks with regular topologies. *IEEE/ACM Transactions on Networking 7*, 2 (1999), 241–250.

[40] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. *A scalable content-addressable network*, vol. 31. ACM, 2001.

[41] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing* (2001), Springer, pp. 329–350.

[42] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the social network's (datacenter) network. In *ACM SIGCOMM* (2015).

[43] SANFILIPPO, S., AND NOORDHUIS, P. Redis. "http://redis.io", 2010.

[44] SCHRIJVER, A. Bipartite edge-colouring in o(δm) time. *SIAM Journal on Computing* (1998).

[45] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The hadoop distributed file system. In *IEEE MSST* (2010).

[46] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., ET AL. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *ACM SIGCOMM* (2015).

[47] SKOMOROCH, P. Wikipedia traffic statistics dataset. http://aws.amazon.com/datasets/2596, 2009.

[48] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for

internet applications. *ACM SIGCOMM Computer Communication Review 31*, 4 (2001), 149–160.

[49] VASSEUR, J.-P., PICKAVET, M., AND DEMEESTER, P. *Network recovery: Protection and Restoration of Optical, SONET-SDH, IP, and MPLS*. Elsevier, 2004.

[50] WANG, G., ANDERSEN, D., KAMINSKY, M., PAPAGIANNAKI, K., NG, T., KOZUCH, M., AND RYAN, M. c-Through: Part-time optics in data centers. In *ACM SIGCOMM* (2010).

[51] WANG, H., XIA, Y., BERGMAN, K., NG, T., SAHU, S., AND SRIPANIDKULCHAI, K. Rethinking the physical layer of data center networks of the next decade: Using optics to enable efficient*-cast connectivity. In *ACM SIGCOMM Computer Communication Review* (2013).

[52] WON, R., AND PANICCIA, M. Integrating silicon photonics, 2010.

[53] YAMAMOTO, N., AKAHANE, K., KAWANISHI, T., KATOUF, R., AND SOTOBAYASHI, H. Quantum dot optical frequency comb laser with mode-selection technique for 1-μm waveband photonic transport system. In *Japanese Journal of Applied Physics* (2010).

[54] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX NSDI* (2012).

[55] ZHU, Z. Scalable and topology adaptive intra-data center networking enabled by wavelength selective switching. In *OSA/OFC/NFOFC* (2014).

[56] ZHU, Z., ZHONG, S., CHEN, L., AND CHEN, K. A fully programmable and scalable optical switching fabric for petabyte data center. In *Optics Express* (2015).

Appendix

Proof for non-blocking connectivity

We first formulate the wavelength assignment problem:

Problem formulation: Denote $G{=}(V{,}E)$ as a MegaSwitch logical graph, where V/E are node/edge sets, and each edge $e{\in}E$ is directed (a pair of fiber channels between two nodes, one in each direction). Given all nodes are connected via one hop, G is a complete digraph. The bandwidth demand between nodes can be represented by the number of wavelengths needed, and each node has k available wavelengths. Suppose $\gamma{=}\{k_e \mid e{\in}E\}$ is a bandwidth demand of a communication, where k_e is the number of wavelengths (i.e., bandwidth) needed on edge $e{=}(u{,}v)$ from node u to v. A feasible demand and wavelength efficiency are defined as:

Definition 2. *A demand γ is feasible iff $\sum_v k_{(u,v)} \leq k, \forall u{\in}V$ and $\sum_u k_{(u,v)} \leq k, \forall v{\in}V$, i.e. each node cannot send/receive more than k wavelengths.*

Definition 3. *Given a feasible bandwidth demand γ, an assignment is wavelength efficient iff it is non-interfering, satisfies γ, and uses at most k wavelengths.*

Centralized optimal algorithm: To prove the wavelength efficiency, we first show that non-interfering wavelength assignment in MegaSwitch can be recast as an edge-coloring problem on a bipartite multigraph. We first transform G into a directed bipartite graph by decomposing each node in G into two logical nodes s_i (sources) and d_i (destinations). Each edge points from a node in $\{s_i\}$ to a node in $\{d_i\}$. Then, we expand G to a multigraph G' by duplicating each edge e between two nodes k_e times. On this multigraph, the requirement of non-interference is that no two adjacent edges share same wavelength. Suppose each wavelength has a unique color, this equivalently transforms to the edge-coloring problem.

While the edge-coloring problem is $\mathcal{NP}$-hard for general graph [33], polynomial-time optimal solutions exist for bipartite multigraph. Chen et al. [7] have presented such a centralized algorithm, which we leverage to prove our wavelength efficiency for any feasible demands.

Algorithm 1: DECOMPOSE$(\cdot)$ Decompose G'_r into $\Delta(G'_r)$ perfect matchings

Input: G'_r
Output: $\pi{=}\{m_1{,}m_2{,}...m_{\Delta(G'_r)}\}$

```
if Δ(G'_r)≤1 then
    return G'_r;
else
    m ← Perfect_Matching(G'_r);
    return m ⋃ Decompose(G'_r \ m);
```

Satisfying arbitrary feasible γ with wavelength efficiency: We extend G' to a $\Delta(G')$-regular multigraph[16], G'_r, by adding dummy edges, where $\Delta(G')$ is the largest vertex degree of G', *i.e.* the largest k_e in γ. For a bipartite graph G' with maximum degree $\Delta(G')$, at least $\Delta(G')$ wavelengths are needed to satisfy γ. This optimality can be achieved by showing that we only need the smallest value possible, $\Delta(G')$, to satisfy γ, which also proves wavelength efficiency since $\Delta(G'){\leq}k$.

Theorem 4. *A feasible demand γ only needs $\Delta(G')$, i.e., the minimum number of colors, for edge-coloring. Any feasible MegaSwitch graph G' can be satisfied with $\Delta(G'){\leq}k$ wavelengths using Algorithm 1.*

Proof. DECOMPOSE$(\cdot)$ recursively finds $\Delta(G^r)$ perfect matchings in G'_r, because "any k-regular bipartite graph has a perfect matching" [44]. Thus, we can extract one perfect matching from the original graph G'_r, and the residual graph becomes $(\Delta(G'){-}1)$-regular; we continue to extract one by one until we have $\Delta(G')$ perfect matchings[17]. Thus, any demand described by G' can be satisfied with $\Delta(G')$ wavelengths. Note that $\Delta(G'){\leq}k$, as the out-degree and in-degree of any node must be $\leq k$ in any feasible demand due to the physical limitation of k ports. Therefore any feasible demand γ can be satisfied using at most k wavelengths. □

The MegaSwitch controller runs Algorithm 1 to decompose the demands, and assigns a wavelength to the matching generated in each iteration.

Considerations for Basemesh Topology

As described in §3.2.2, basemesh is an overlay DHT network on the multi-fiber ring of MegaSwitch. DHT literature is vast, and there are many potential choices for basemesh topology. The following are the representative ones: Chord [48], Pastry [41], Symphony [29], Viceroy [28], and CAN [40]. Since we have compared Chord [48] & Symphony [29] in §3.2.2, we next look at the remaining ones.

- Pastry suffers from average path lengths when many wavelengths are used in the basemesh. Its average path length is $log_2(l{\cdot}n)$ [41], where n is the number of nodes on a ring, and l is the length of node ID in bits. For MegaSwitch, both parameter is fixed (like Chord), and we cannot reduce the path length by adding more wavelengths to basemesh.
- Viceroy emulates the butterfly network on a DHT ring. For MegaSwitch, its main issue is the difficulty of updating the routing tables and wavelength assignments

[16] A graph is regular when every vertex has the same degree.

[17] Perfect_Matching$(\cdot)$ on regular bipartite graph is well studied, we leverage existing algorithms in literature [27].

when the capacity of basemesh is increased and decreased. For Symphony, we can just pick a new wavelength in random, and configure accordingly without affecting the other wavelengths. In contrast, to maintain butterfly topology, adjusting capacity of basemesh using Viceroy algorithm affects all wavelength but one in the worst case.

- CAN is a d-dimensional Cartesian coordinate system on a d-torus, which is not suitable for MegaSwitch ring where every node is connected directly to every other node. However, CAN will become useful when we expand MegaSwitch into a 2-D torus topology, and we will explore this as future work.

Reconfiguration Latency Measurements on MegaSwitch Prototype

In §5.1.1, we measured ∼20ms reconfiguration delay for MegaSwitch prototype, and here we break down this delay into EPS configuration delay (t_e), WSS switching delay (t_o), and control plane delay (t_c). The total reconfiguration delay for MegaSwitch is t_r=max(t_e,t_o)+t_c, as EPS and WSS configurations can be done in parallel.

EPS configuration delay: To setup a route in MegaSwitch, the source/destination EPSes must be configured. We measure the EPS configuration delay as follows (all servers and the controller are synchronized by NTP): we first setup a wavelength between 2 nodes and leave EPSes unconfigured. We let a host in one of the nodes keep generating UDP packets using `netcat` to a host in the other node. Then, the controller sends OpenFlow control message to both EPSes to setup the route, and we collect the packets at the receiver with `tcpdump`. Finally we can calculate the delay: the average and 99th percentile are 5.12ms and 12.87ms, respectively.

WSS switching delay: We measure the switching speed of the 8×1 WSS used in our testbed by switching the optical signal from one port to another port. Figure 12 shows the power readings from the original port and from the destination port, and the measured switching delay is 3.23ms (subject to temperature, drive voltage, etc.).

Control plane delay: With our wavelength assignment algorithm (see Appendix) running on a server (the centralized controller) with Intel E5-1410 2.8Ghz CPU, we measured an average computation time of 0.53ms for 40-port (40×40 demand matrix as input), and 3.28ms for 6336-port MegaSwitch (n=33,k=192). For basemesh routing tables, our greedy algorithm runs 0.45ms for 40 ports and 1.78ms for 6336 ports. For the OWS controller processing, we measured 4.31ms from receiving a wavelength assignment to set GPIO outputs. Round-trip time in control plane network (using 1GbE switch) is 0.078ms.

Passive Realtime Datacenter Fault Detection and Localization

Arjun Roy, Hongyi Zeng†, Jasmeet Bagga†, and Alex C. Snoeren

UC San Diego and †Facebook, Inc.

ABSTRACT

Datacenters are characterized by their large scale, stringent reliability requirements, and significant application diversity. However, the realities of employing hardware with small but non-zero failure rates mean that datacenters are subject to significant numbers of failures, impacting the performance of the services that rely on them. To make matters worse, these failures are not always obvious; network switches and links can fail partially, dropping or delaying various subsets of packets without necessarily delivering a clear signal that they are faulty. Thus, traditional fault detection techniques involving end-host or router-based statistics can fall short in their ability to identify these errors.

We describe how to expedite the process of detecting and localizing partial datacenter faults using an end-host method generalizable to most datacenter applications. In particular, we correlate transport-layer flow metrics and network-I/O system call delay at end hosts with the path that traffic takes through the datacenter and apply statistical analysis techniques to identify outliers and localize the faulty link and/or switch(es). We evaluate our approach in a production Facebook front-end datacenter.

1. INTRODUCTION

Modern datacenters continue to increase in scale, speed, and complexity. As these massive computing infrastructures expand—to hundreds of thousands of multi-core servers with 10- and 40-Gbps NICs [35] and beyond—so too do the sets of applications they support: Google recently disclosed that their datacenter network fabrics support literally thousands of distinct applications and services [37]. Yet the practicality of operating such multi-purpose datacenters depends on effective management. While any given service might employ an army of support engineers to ensure its efficient operation, these efforts can be frustrated by the inevitable failures that arise within the network fabric itself.

Unfortunately, experience indicates that modern datacenters are rife with hardware and software failures—indeed, they are designed to be robust to large numbers of such faults. The large scale of deployment both ensures a non-trivial fault incidence rate and complicates the localization of these faults. Recently, authors from Microsoft described [44] a rogue's gallery of datacenter faults: dusty fiber-optic connectors leading to corrupted packets, switch software bugs, hardware faults, incorrect ECMP load balancing, untrustworthy counters, and more. Confounding the issue is the fact that failures can be intermittent and partial: rather than failing completely, a link or switch might only affect a subset of traffic, complicating detection and diagnosis. Moreover, these failures can have significant impact on application performance. For example, the authors of NetPilot [42] describe how a single link dropping a small percentage of packets, combined with cut-through routing, resulted in degraded application performance and a multiple-hour network goose chase to identify the faulty device.

Existing production methods for detecting and localizing datacenter network faults typically involve watching for anomalous network events (for example, scanning switch queue drop and link utility/error counters) and/or monitoring performance metrics at end hosts. Such methods consider each event independently: "Did a drop happen at this link? Is application RPC latency unusually high at this host?" Yet, in isolation, knowledge of these events is of limited utility. There are many reasons an end host could observe poor network performance; similarly, in-network packet drops may be the result of transient congestion rather than a persistent network fault. Hence, datacenter operators frequently fall back to active probing and a certain degree of manual analysis to diagnose and localize detected performance anomalies [19, 44].

Instead, we propose an alternative approach: rather than looking at anomalies independently, we consider the impacts of faults on aggregate application performance. Modern datacenter fabrics are designed with a plethora of disjoint paths and operators work hard to load balance traffic across both paths and servers [9, 37]. Such designs result in highly regular flow performance regardless of path—in the absence of network faults [35]. An (unmitigated) fault, on the other hand, will manifest itself as a performance anomaly visible to end hosts. Hence, to detect faults, we can compare performance end hosts observe along different paths and hypothesize that outliers correspond to faults within the network.

To facilitate such a comparison, we develop a lightweight packet-marking technique—leveraging only forwarding rules supported by commodity switching ASICs—that uniquely identifies the full path that a packet traverses in a Facebook datacenter. Moreover,

the topological regularity of Facebook's datacenter networks allows us to use path information to passively localize the fault as well. Because each end host can bin flows according to the individual network elements they traverse, we can contrast flows traversing any given link (switch) at a particular level in the hierarchy with flows that traverse alternatives in order to identify the likely source of detected performance anomalies. Operators can then use the output of our system—namely the set of impacted traffic and the network element(s) seemingly responsible—in order to adjust path selection (e.g., through OpenFlow rules, ECMP weight adjustment [32], or tweaking inputs to flow hashes [23]) to mitigate the performance impact of the fault until they can repair it.

A naive implementation of our approach is unlikely to succeed given the noisiness of flow-based metrics at individual host scale. Furthermore, distinct applications, or different workloads for the same application, are likely to be impacted differently by any given fault. Here the massive scale of modern datacenters aids us: Specifically, when aggregated across the full set of traffic traversing any given network link, we find that statistical techniques are effective at using end-host-based metrics to identify under-performing links in real time. Our experience suggests this can remain true even when end hosts service different requests, communicate with disjoint sets of remote servers, or run entirely distinct application services.

To be practical, our approach must not place a heavy burden on the computational resources of either end hosts or the network switches, and neither can we require significant network bandwidth or esoteric features from the switches themselves. Furthermore, our analysis techniques must avoid false positives despite the diversity and scale of production datacenters yet remain sensitive to the myriad possible impacts of real-word faults. Our contributions include (1) a general-purpose, end-host-based performance monitoring scheme that can robustly identify flows traversing faulty network components, (2) a methodology to discover the necessary path information scalably in Facebook's datacenters, and (3) a system that leverages both types of information in aggregate to perform network-wide fault-localization.

At a high level, while network statistics can be noisy and confusing to interpret in isolation, the regular topology and highly engineered traffic present within Facebook's datacenters provides an opportunity to leverage simple statistical methods to rapidly determine where partial faults occur as they happen. We demonstrate that our technique is able to identify links and routers exhibiting low levels (0.25–1.0%) of packet loss within a Facebook datacenter hosting user-servicing front-end web and caching servers within 20 seconds of fault occurrence with a minimal amount of processing overhead. We also perform a sensitivity analysis on a testbed to consider different types of errors—including those that induce only additional latency and not loss—traffic patterns, application mixes, and other confounding factors.

2. MOTIVATION & RELATED WORK

While fault detection is a classical problem in distributed systems [7, 11, 13, 14, 18, 33] and networks [12, 24, 29], modern datacenter environments provide both significant challenges (e.g., volume and diversity of application traffic, path diversity, and stringent latency requirements) and benefits (large degree of control, regular topologies) that impact the task of effectively finding and responding to network faults. Moreover, recent work has indicated that the types and impacts of faults common in modern datacenters [42, 44] differ from those typically encountered in the wide-area [34, 40] and enterprise [39].

Datacenters are affected by a menagerie of errors, including a mix of software errors (ECMP imbalances, protocol bugs, etc.), hardware errors (packet corruption due to cables or switches, unreliable packet counters, bit errors within routing tables, etc.), configuration errors, and a significant number of errors without an apparent cause [44]. Errors can be classed into two main categories: a complete error, in which an entire link or switch is unable to forward packets, or a partial error, where only a subset of traffic is affected. In this work, we focus primarily on the latter. Even a partial error affecting and corrupting only 1% of packets on two links inside a datacenter network was noted to have a $4.5\times$ increase in the 99th-percentile application latency [42], which can impact revenue generated by the hosted services [21].

Commonly deployed network monitoring approaches include end-host monitoring (RPC latency, TCP retransmits, etc.) and switch-based monitoring (drop counters, queue occupancies, etc.). However, such methods can fall short for troubleshooting datacenter scale networks. Host monitoring alone lacks specificity in the presence of large scale multipath; an application suffering from dropped packets or increased latency does not give any intuition on where the fault is located, or whether a given set of performance anomalies are due to the same fault.

Similarly, if a switch drops a packet, the operator is unlikely to know which application's traffic was impacted, or, more importantly, what is to blame. Even if a switch samples dropped packets the operator might not have a clear idea of what traffic was impacted. Due to sampling bias, mouse flows experiencing loss might be missed, despite incurring great impacts to performance. (We expound upon this issue in Appendix A.) Switch-counter-based approaches are further confounded by cut-through forwarding and unreliable hardware [44].

Thus, we propose a system that focuses not on the occurrence of network anomalies but rather on their impact on traffic. In doing so we leverage the high path diver-

sity and regular topologies found in modern datacenters, as well as the finely tuned load balancing common in such environments. We also expose to each host, for every flow, the path taken through the network. As such, our approach has several key differences with both past academic proposals and production systems:

1. **Full path information**: Past fault-finding systems have associated performance degradations with components and logical paths [7, 10, 13, 19] but a solution that correlates performance anomalies with specific network elements for arbitrary applications has, to the best of our knowledge, proved elusive—although solutions exist for carefully chosen subsets of traffic [44].
2. **Passive monitoring, not active probing**: In contrast to active probing methods [17, 19], our method leverages readily available metrics from production traffic, simultaneously decreasing network overhead, increasing the detection surface, and decreasing detection and localization time.
3. **Reduced switch dependencies**: While some approaches require expanded switch ASIC features for debugging networks [20], we do not require them. This allows network operators to deploy our approach on commodity switches, and makes our approach resilient to bugs that may escape on-switch monitoring.
4. **No per-application modeling:** Our system leverages the stable and load-balanced traffic patterns found in some modern datacenters [35]. Thus, it does not need to model complicated application dependencies [7, 11, 13] or interpose on application middleware [14]. Since we compare relative performance across network links, we do not require explicit performance thresholds.
5. **Rapid online analysis**: The regularity inherent in several academic [8] and production [9, 37] topologies allow us to simplify our analysis to the point that we can rapidly (10–20 seconds) detect partial faults of very small magnitude ($\leq$ 0.5% packet loss) using an online approach. This contrasts with prior systems that require offline analysis [7] or much larger timescales to find faults [11]. Furthermore, localizing faults is possible without resource-intensive and potentially time consuming graph analysis [17, 24, 25, 26, 27, 30].

3. SYSTEM OVERVIEW

In this section, we present the high-level design of a system that implements our proposed approach. To set the context for our design, we first outline several important characteristics of Facebook's datacenter environment. We then introduce and describe the high-level responsibilities of the three key components of our system, deferring implementation details to Section 4.

3.1 Production datacenter

Facebook's datacenters consist of thousands of hosts and hundreds of switches grouped into a multi-rooted, multi-level tree topology [9]. The datacenter we consider serves web requests from a multitude of end users, and is comprised primarily of web servers and cache servers [35]. While wide-area connections to other installations exist, we do not focus on those in this work.

3.1.1 Topology

Web and cache hosts are grouped by type into racks, each housing a few tens of hosts. User requests are load balanced across all web servers, while cached objects are spread across all caches. Since any web server can service any user request, there is a large fan-out of connections between web servers and caches; in particular, each web server has thousands of bidirectional flows spread evenly amongst caches [35]. Prior work notes both the prevalence of partition-aggregate workloads and the detrimental impact of packet loss and delay in this latency sensitive environment—even if they only constitute the long tail of the performance curve [43].

A few tens of racks comprises a pod. Each pod also contains four aggregation switches (Aggs). Each ToR has four uplinks, one to each Agg. There are a few tens of pods within the datacenter, with cross-pod communication enabled by four disjoint planes of core switches (each consisting of a few tens of cores). Each Agg is connected to the cores in exactly one plane in a mesh.

Each host transmits many flows with differing network paths. Due to the effects of ECMP routing, mesh-like traffic patterns, and extensive load balancing, links at the same hierarchical level of the topology end up with a very even distribution of a large number of flows. Moreover, if we know which path (i.e., set of links) every flow traverses, it is straightforward to separate the flows into bins based on the link they traverse at any particular level of the hierarchy. Hence, we can simultaneously perform fault identification and localization by considering performance metrics across different subsets of flows.

3.1.2 Operational constraints

The sheer scale of the datacenter environment imposes some significant challenges on our system. The large number of links and switches require our system to be robust to the presence of multiple simultaneous errors, both unrelated (separate components) and correlated (faults impacting multiple links). While some errors might be of larger magnitude than others, we must still be sensitive to the existence and location of the smaller errors. In addition, we must detect both packet loss and delay.

The variety of applications and workloads within the datacenter further complicate matters—an improper choice of metric can risk either masking faults or triggering false positives (for example, the reasonable sounding choice of request latency is impacted not only by net-

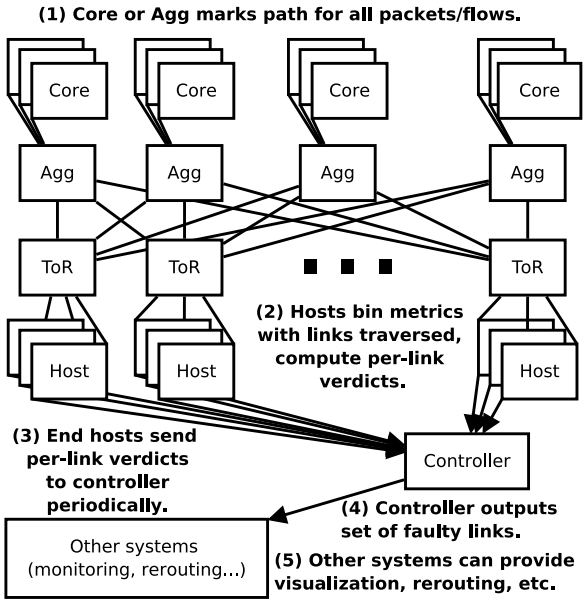


Figure 1: High-level system overview (single pod depicted).

work faults but also cache misses, request size and server loads). Moreover, datacenters supporting multiple tenants clearly require application-agnostic metrics.

Furthermore, we must be able to support measurements from large numbers of end hosts describing the health of a large number of links, without imposing large computational or data overheads either on the end hosts or on the network. This is especially true of network switches, where relatively under-provisioned control planes are already engaged in critical tasks including BGP routing. Thus, we are limited to capabilities present in the data planes of commodity switch ASICs.

3.2 System architecture

Our fault detection and localization approach involves functional components at all end hosts, a subset of switches, and a centralized controller, depicted in Figure 1. Switches mark packets to indicate network path (1). Hosts then independently compare the performance of their own flows to generate a host-local decision about the health of all network components (2). These *verdicts* are sent (3) to a central controller, which filters false positives to arrive at a final set of faulty components (4), which may be further acted upon by other systems (5). We expand on the role of each element below.

3.2.1 End hosts

Hosts run production application traffic and track various per-flow metrics detailed in Section 4.2. In addition, the host is aware of each flow's path through the network. Periodically, hosts will use collected performance data to issue verdicts for whether it considers a given subset of flows to have degraded performance, or not. By default, flow metrics are binned by the set of links they traverse. These bins are then further grouped into what we call *equivalence sets* (ESes), i.e., the set of bins that should perform equivalently, allowing us to pinpoint link-level faults. In the Facebook datacenter, the set of bins corresponding to the downlinks from the network core into a pod forms one such ES. Alternative schemes can give us further resolution into the details of a fault: for example, comparing traffic by queue or subnet (Section 5.4.4). We discuss the impact of heterogeneous traffic and topologies on our ability to form ESes in Section 6.

We define a *guilty* verdict as an indication that a particular bin has degraded performance compared to others in its ES; a *not guilty* verdict signifies typical performance. We leverage path diversity and the ability to compare performance across links—if every link in an ES is performing similarly, then either none of the links are faulty, all of them are faulty (unlikely in a production network) or a fault exists but might be masked by some other bottleneck (for which we cannot account). The target case, though, is that enough links in an ES will be fault-free at any given moment, such that the subset of links experiencing a fault will be readily visible if we can correlate network performance with link traversed. Even in the absence of path diversity (e.g., the access link for a host) we can use our method with alternative binning schemes and equivalence sets to diagnose certain granular errors.

3.2.2 Switches

A subset of the network switches are responsible for signaling to the end hosts the network path for each flow; we describe details in Section 4.1. Once faults are discovered by the centralized controller, switches could route flows away from faulty components, relying on the excess capacity typically found in datacenter networks. We leave fault mitigation to future work.

3.2.3 Controller

In practice, there will be some number of false positives within host-generated verdicts for link health (e.g. hosts flagging a link that performs poorly even in the absence of an error, possibly due to momentary congestion) confounding the ability to accurately deliver fixes to the network. Furthermore, there might be errors which do not affect all traffic equally; for example, only traffic from a certain subnet might be impacted, or traffic of a certain class. Hence, we employ a central controller that aggregates end-host verdicts for links (or other bins) into a single determination of which links—if any—are suffering a fault. In addition, the controller can drive other administrative systems, such as those used for data visualization/logging or rerouting of traffic around faults. Such systems are outside the scope of this work.

4. IMPLEMENTATION

We now present a proof-of-concept implementation that meets the constraints presented above. In order, we focus on scalable path signalling, our choice of end-host

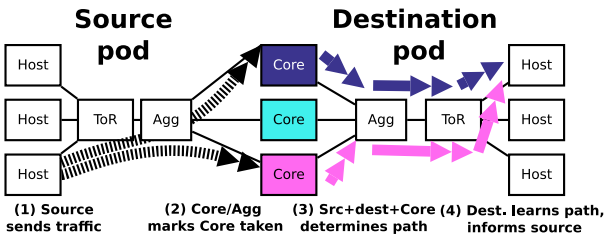


Figure 2: Determining flow network path.

performance metrics and the required aggregation processing, our verdict generator for aggregated flow metrics, and the operation of our centralized controller.

4.1 Datacenter flow path discovery

Central to our approach is the ability to scalably and feasibly discover flow path information within the datacenter. While switch CPU/dataplane limits complicate this task, the regularity of the network topology aids us.

4.1.1 Topological constraints

Figure 2 depicts the pathfinding scheme we use in the Facebook datacenter. To aid discussion, the diagram shows a restricted subset of an unfolded version of the topology, with source hosts on the left and destinations on the right; the topology is symmetric so our approach works in either direction. Note that cross-pod traffic has only two ECMP decisions to make: which Agg switch (and thus, core plane) to transit after the ToR, and which core switch to use within the core plane. Each core plane is only connected to one Agg switch per pod, so the destination Agg switch is fixed given the core plane.

The source and destination racks and ToRs are fixed for any particular host pair, and can be determined by examining the IP addresses of a flow. Thus, for cross-pod traffic, the choice of core switch uniquely determines the flow's path, as the Agg switches are then constrained on both sides. For intra-pod traffic it suffices to identify the Agg switch used to connect the ToRs.

In the presence of a total link error, the network attempts to forward traffic using an alternative, non-shortest path advertised as a backup route. While we do not address this case in our proof of concept, we discuss the impacts of link failures in Section 6.3.

4.1.2 Packet marking

We assign an ID to each core switch, that is stamped on all packets traversing the switch. Note that the stamp need not be inserted by the core switch itself—the Agg switches on either side are also aware of the core's identity and are equally capable of marking the packet. We use Linux eBPF (Extended Berkeley Packet Filter) [3] along with `bcc` (BPF Compiler Collection) [2] instrumentation at end hosts to read packet markings and derive flow paths. Our naive implementation imposes less than 1% CPU overhead (top row of Table 2), but room for optimization remains.

Several candidate header fields can be used for marking, and the best choice likely depends on the details of any given deployment. One possibility is the IPv6 flow label field; a 20-bit ID could scale to a network with over a million core switches. However, the ASIC in our Agg switches does not currently support modifying this field. Thus, for our proof of concept, we instead mark the IPv6 DSCP field, which is supported at line rate and requires only a constant number of rules (one per uplink).

While DSCP suffices for a proof of concept, its length limits the number of discernible paths. Furthermore, datacenter operators often use DSCP to influence queuing, limiting the available bits. One alternative is to mark the TTL field[1]. A packet marked at a downward-facing Agg traverses exactly two more hops before arriving at the destination host; a host could recover an ID in the TTL field as long as the value was in the range 3–255.

4.1.3 General case

More complex topologies might preclude our ability to compute paths from a single stamp. In the event that traffic is made up of long-lived flows (as it is, for example, for the web and cache servers we tested) we can leverage the match/mark capability that many switching ASICs possess to implement a form of marking reminiscent of previous proposals for IP traceback [36].

Suppose there are H hops between source and destination system, and C routing choices per hop. If we want to determine the path taken by a flow at the first hop where a routing choice must be made, the operating system can mark C packets, each with the IPv6 flow label set to the possible IDs of each switch that the packets could transit for that hop. Switches would be configured with a single rule that would examine the flow label—if it matches the switch ID, the switch would set a single DSCP bit to 1. When the end host receiving the packet notes the DSCP bit set to 1, it could signal the sender with the ID of the switch that was transited at that hop (that is, the flow label of the packet when the DSCP bit was set). For example, it could do this by setting a specific DSCP bit in the ACK packet while setting the flow label of the return packet to the switches ID. Thus, if the flow sends $\geq (H \cdot C)$ packets in total it can discover the entire path, at the expense of just a single rule per switch. While we have not deployed this approach in production, we validated that our switch hardware can implement it. While this method finds out path information hop by hop, partial path information can still be useful to our system. We discuss this further in Section 6.4.

4.2 Aggregating host metrics & path data

Given the availability of per-flow path information, we show that both transport-layer and system-call-timing

[1]Constraints exist on the use of TTL as well, such as in the presence of `traceroute` or eBGP session protection [15].

metrics can be used to find under-performing links in real-world scenarios. We consider both latency-sensitive, client/server production applications [35] and bulk-flow, large-scale computation applications [4, 16, 22]. Mixed traffic is considered in Section 6.2.

4.2.1 Latency sensitive services

Web servers and caches within the Facebook datacenter service user requests, where low latency is desirable [43]. Faults harm performance, where either drops or queuing delay can result in unacceptable increases in request latency. Since loss is often a sign of network congestion, the TCP state machine tracks various related statistics. These include the number of retransmitted packets, the congestion window (`cwnd`) and the slow-start threshold (`sstresh`). Latency is also tracked using smoothed round trip time (`srtt`). When considered in isolation, these metrics are limited in usefulness; while a retransmit signifies diminished performance for a flow, it does not provide any predictive power for the underlying cause, or whether a given set of flows experiencing retransmits are doing so for the same underlying reason. Furthermore, while `cwnd` and `ssthresh` decrease with loss, the specific values depend highly on the application pattern. For example, bulk flows tend to have a larger congestion window than mouse flows. Thus, comparing any given flow against an average can be difficult, since it is unclear whether 'worse' performance than average is due to a network issue or traffic characteristics.

However, when augmented with path data, these metrics can become valuable indicators of faults. To illustrate the effectiveness of this approach, we induced a network fault impacting two hosts: one web server, and one cache server. In each case, the local ToR is connected to four aggregation switches. Using `iptables` rules, each host dropped 0.5% of incoming packets that transited the link from the first aggregation switch to the ToR switch. We then measured the TCP metrics for outgoing flows, grouped by which aggregation switch (and, thus, rack downlink) was transited on the way to the host.

Rows 1–6 and 9–14 in Table 1 depict the `cwnd`, `ssthresh`, and retransmission count distributions grouped by inbound downlink for production cache (C) and web servers (W) respectively. While we aggregate the values for the non-faulty links into a single series (the even rows in the table), each individual non-faulty link follows the same aggregate distribution with little variation. On the other hand, the faulty-link distribution for each metric is significantly skewed—towards smaller numbers for `cwnd` and `ssthresh`, and larger for retransmits. Rows 17–22 show that `cwnd`, `ssthresh`, and retransmits provide a similarly strong signal when the link fault impacts traffic in the outbound direction instead. `srtt` is effective for detecting faults that induce latency but not loss; we defer details to Section 5.3.1.

4.2.2 Bulk data processing

Next, we consider bulk data processing workloads. Commonly used frameworks like Hadoop involve reading large volumes of data from various portions of the network; slowdowns caused by the network can have a disproportionate impact on job completion times due to stragglers. While the TCP metrics above work equally well in the case of Hadoop (not shown), the high-volume flows in Hadoop allow us to adopt a higher-level, protocol-independent metric that depends on the buffer dynamics present in any reliable transport protocol.

Consider the case of an application making a blocking system call to send data. Either there will be room present in the connection's network buffer, in which case the data will be buffered immediately, or the buffer does not have enough space and causes the application to wait. As packets are transmitted, the buffer is drained. However, if a fault induces packet drops, packets need to be retransmitted and thus the goodput of the network buffer drops. Correspondingly, the buffer stays full more of the time and the distribution of the latency of `send()` and similar blocking system calls skews larger. Delays caused by packet reordering have similar impacts. Non-blocking sends exhibit this behavior too; we can instrument either the `select()` or `epoll_ctl()` system calls to get insight into buffer behavior.

To demonstrate this, we consider a synthetic traffic pattern representative of the published flow-size distributions for Hadoop workloads present in Facebook's datacenters [35]. Specifically, we designate one server in our testbed (see Section 5.1 for details) as a sink and the remainder as sources. In addition, we induce a random packet drop impacting 1 of 9 testbed core switches. For a fixed period of time, each server in a loop creates a fixed number of simultaneous sender processes. Each sender starts a new flow to the sink, picks a flow size from the Facebook flow-size distribution for Hadoop servers and transmits the flow to completion, while recording the wait times for each `select()` system call.

Rows 23–24 in Table 1 show the distributions of `select()` latencies for flows grouped by core switch transited in our private testbed—again, the non-faulty distributions are aggregated. Faulty links on the impacted core yield a dramatically shifted distribution. Additionally, the distributions for non-faulty cores show little variation (omitted for space). This allows us to differentiate between faulty and normal links and devices. We found the metric to be sensitive to drop rates as low as 0.5%. Moreover, this signal can also be used for caches, due to the long-lived nature of their flows. Rows 7–8 and 15–16 in Table 1 show the distribution of `epoll()` and `select()` latencies for flows on faulty and non-faulty links in production and on the testbed, respectively; in both cases, the faulty link distribution skews larger.

#	Host	Type	Metric	Path	Error	p25	p50	p75	p90	p95	p99
1	(C)	Prod	cwnd	In	0.5%	7 (-30%)	8 (-20%)	10 (par)	10 (par)	10 (-41%)	20 (-33%)
2	(C)	Prod	cwnd	In	-	10	10	10	10	17	30
3	(C)	Prod	ssthresh	In	0.5%	6 (-62.5%)	7 (-63.2%)	18 (-25%)	24 (-64.7%)	31 (-51.6%)	63 (-17.1%)
4	(C)	Prod	ssthresh	In	-	16	19	24	57	64	76
5	(C)	Prod	retx	In	0.5%	0 (par)	1	2	3	4	6
6	(C)	Prod	retx	In	-	0	0	0	0	0	1
7	(C)	Prod	epoll	In	0.5%	0.003 (par)	0.14 (+1.4%)	0.47(+10.8%)	0.71(+30.6%)	1.07 (+60.6%)	2.28 (+125%)
8	(C)	Prod	epoll	In	-	0.003	0.14	0.43	0.54	0.67	1.01
9	(W)	Prod	cwnd	In	0.5%	8 (-63.6%)	12 (-60%)	17 (-74.6%)	38 (-60%)	121 (+23.5%)	139 (-33.2%)
10	(W)	Prod	cwnd	In	-	22	30	67	95	98	208
11	(W)	Prod	ssthresh	In	0.5%	4 (-42.9%)	12 (-40%)	16 (-66.7%)	19 (-75%)	31 (-66.7%)	117 (+19.4%)
12	(W)	Prod	ssthresh	In	-	7	20	48	73	93	98
13	(W)	Prod	retx	In	0.5%	0	0	1	3	4	6
14	(W)	Prod	retx	In	-	0	0	0	0	0	0
15	(C)	Syn	select	Out	2.0%	0.56(+25k%)	4.22(+717%)	7.01(+642%)	37.5(+1.6k%)	216 (+4.3k%)	423 (+969%)
16	(C)	Syn	select	Out	-	0.002	0.516	0.944	2.15	4.90	39.6
17	(W)	Syn	cwnd	Out	0.5%	2 (-80%)	2 (-80%)	2 (-80%)	4 (-60%)	7 (-30%)	10 (par)
18	(W)	Syn	cwnd	Out	-	10	10	10	10	10	10
19	(W)	Syn	ssthresh	Out	0.5%	2 (-50%)	2 (-71%)	2 (-75%)	2 (-78%)	4 (-56%)	7 (-22%)
20	(W)	Syn	ssthresh	Out	-	4	7	8	9	9	9
21	(W)	Syn	retx	Out	0.5	21	23	26	29	30	40
22	(W)	Syn	retx	Out	-	0	0	0	0	0	0
23	(H)	Syn	select	Out	2.0%	22 (+11%)	223 (+723%)	434 (+85%)	838 (+32%)	1244 (+47%)	2410 (+39%)
24	(H)	Syn	select	Out	-	19.5	27.1	235	634	844	1740

Table 1: Metric statistics for Production/Synthetic (C)ache/(W)eb/(H)adoop hosts grouped by (In/Out)bound path and induced loss rates; sycall metrics in msec. Each color-banded pair of rows denotes the base and impacted metrics for (aggregated) working and (unique) faulty paths.

Component	p25	p50	p75	p95
eBPF (paths)	0.17%	0.23%	0.46%	0.65%
TCP metrics/t-test	0.25%	0.27%	0.29%	0.33%

Table 2: End-host monitoring CPU utilization in production.

4.2.3 Metric collection computational overhead

While the aforementioned statistics provide a useful signal, care must be taken when determining how to collect statistics. While system-call latency can be instrumented within an application, that requires potentially invasive code changes. Instead, we again leverage eBPF to track system-call latencies. For TCP statistics, we directly read `netlink` sockets in a manner similar to the `ss` command. Table 2 depicts the overall CPU usage of our TCP statistics collection and verdict generation agent; the cumulative CPU overhead is below 1%.

4.3 Verdict generation

Previously, we demonstrated that path aggregated end-host metrics can single out under-performing groups of flows. Here, we develop an end-host decision engine to generate verdicts based on these metrics.

4.3.1 Outlier detection

Due to the clumping of distributions in the absence of faults, we hypothesize that every observed value of a given metric under test can be treated as a sample from the same underlying distribution characterizing a fault-free link, during a particular time period containing a given load. Moreover, a substantially different distribution applies for a faulty link for the same time period. Thus, determining whether a link is faulty reduces to determining whether the samples collected on that link are part of the same distribution as the fault-free links.

While we cannot state with certainty the parameters of the fault-free distribution due to the complexities of network interactions, we rely on the assumption that the number of faulty datacenter links at any given instant is much lower than the number of working links. Suppose we consider the number of retransmits per flow per link, for all links, over a fixed time period. For each link, we compare its distribution to the aggregate distribution for all the other links. If there exists only one fault in the network, then there are two possible cases: the distribution under consideration is faulty and the aggregate contains samples from exclusively non-faulty links, or it is non-faulty and the aggregate contains samples from 1 faulty link and $(N-1)$ working links. In the former case, the distribution under test is skewed significantly to the right of the aggregate; in the latter, it is skewed slightly to the left (due to the influence of the single faulty link in the aggregate). Thus a boolean classifier can be used: if the test distribution is skewed to the right by a sufficiently large amount, we issue a guilty verdict; else, we do not. Concurrent errors shift the aggregate distribution closer to the outlier distribution for a single faulty link. If every link is faulty, we cannot detect faulty links since we do not find any outliers. However, our method is robust even in the case where 75% of the links have faults; we examine the sensitivity of our approach in Section 5.4.2.

4.3.2 Hypothesis testing

Due to the large volume of traffic, even small time intervals contain a large number of samples. Thus, we have to carefully implement our outlier test to avoid significant end-host computational overhead. Our approach

computes a lightweight statistic for each metric under consideration, for each link, over successive fixed-length (10 seconds by default) sampling periods. Each end host generates verdicts for links associated with its own pod. Thus, for each flow, there are a few tens of possible core-to-aggregation downlinks, and four possible aggregation-to-ToR downlinks. Having acquired TCP metrics and path information for all flows via our collection agent, we bucket each sample for the considered metrics into per-link, per-direction buckets. Thus, each metric is bucketed four times: into the inbound and outbound rack and aggregation (up/down)links traversed.

For TCP retransmit and system-call latency metrics, we compute the t-statistic from the single-tailed 1-sample Student's t-test using a low-overhead and constant-space streaming algorithm. To do so we need the average and standard deviation per distribution, and the aggregate average—all of which are amenable to streaming computation via accumulators, including standard deviation via Welford's Algorithm [41]. We then input the t-statistic to the test. The t-test compares a sample mean to a population mean, rejecting the null hypothesis if the sample mean is larger—in our case, if the tested link has more retransmits or higher system-call latency than the other links in aggregate. In every time period, each host checks if the t-statistic is greater than 0 and the p-value ≤ 0.05. If so, we reject the null hypothesis, considering the link to be faulty.

This approach limits the computational overhead at any individual host since it only needs to track statistics for its own `select()/epoll()` calls and TCP statistics. The bottom row of Table 2 depicts the CPU utilization at a single production web server for this approach over 12 hours. The t-test computation and `netlink` TCP statistics reader uses roughly 0.25% of CPU usage. This result is roughly independent of how often the verdict is computed; the majority of the CPU usage is incrementing the various accumulators that track the components of the t-statistic. The `bcc`/eBPF portion, however, has periods of relatively high CPU usage approaching 1% overall, due to the need to periodically flush fixed-sized kernel structures that track flow data.

For `cwnd`, `ssthresh` and `srtt` TCP statistics, we find the student's t-test to be too sensitive in our environment. However, a modified 2-sample Kolmogorov-Smirnov (KS-2) test provides an effective alternative. Specifically, we compare two down-sampled distributions: the 99-point {p1, p2, ..., p99} empirical distribution for the link under consideration, and a similarly defined distribution for the other links in aggregate.

4.4 Centralized fault localization

While individual hosts can issue verdicts regarding link health, doing so admits significant false positives. Instead, we collate host verdicts at a centralized controller that attempts to filter out individual false positives to arrive at a network-wide consensus on faulty links.

4.4.1 Controller processing

Assuming a reliable but noisy end-host-level signal, we hypothesize that false positives should be evenly distributed amongst links in the absence of faults. Note that while some networks might contain hotspots that skew flow metrics and violate this assumption, traffic in the Facebook datacenter is evenly distributed on the considered timescales, with considerable capacity headroom.

We use a centralized controller to determine if all links have approximately the same number of guilty (or not guilty) verdicts, corresponding to the no-faulty-links case. Hosts write link verdicts—generated once per link every ten seconds—to an existing publish-subscribe (pub-sub) framework used for aggregating log data. The controller reads from the pub-sub feed and counts the number of guilty verdicts per link from all the hosts, over a fixed accumulation period (10 seconds by default). The controller flags a link as faulty if it is a sufficiently large outlier. We use a chi-squared test with the null hypothesis that, in the absence of faults, all links will have relatively similar numbers of hosts that flag it not-guilty. The chi-square test outputs a p-value; if it is ≤ 0.05, we flag the link with the least not-guilty verdicts as faulty. We iteratively run the test on the remaining links to uncover additional faults until there are no more outliers.

4.4.2 Computational overhead

The controller has low CPU overhead: a Python implementation computes 10,000 rounds for tens of links in <1 second on a Xeon E5-2660, and scales linearly in the number of links. Each host generates two verdicts per link (inbound/outbound) every 10 seconds, with O(1000s) hosts per pod. Each verdict consists of two 64-bit doubles (t-stat and p-value) and a link ID. In total, this yields a streaming overhead of < 10 Mbps per pod, well within the capabilities of the pub-sub framework.

5. EVALUATION

We now evaluate our approach within two environments: the Facebook datacenter described in Section 3.1, and a small private testbed. First, we describe our test scenario in each network, and provide a motivating example for real-world fault detection. We then consider the speed, sensitivity, precision, and accuracy of our approach, and conclude with experiences from a small-scale, limited-period deployment at Facebook.

5.1 Test environment

Within one of Facebook's datacenters, we instrumented 86 web servers spread across three racks with the monitoring infrastructure described in Section 4. Path markings are provided by a single Agg switch, which sets DSCP bits based on the core switch from which the packet arrived. (Hence, all experiments are restricted to

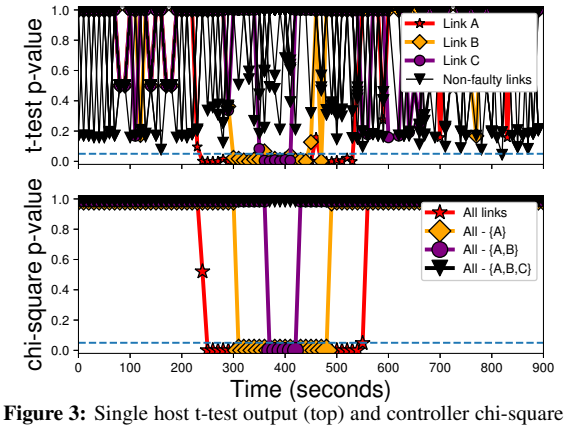


Figure 3: Single host t-test output (top) and controller chi-square output (bottom) for three separate link faults.

the subset of traffic that transits the instrumented Agg switch, and ignores the remainder.) To inject faults, we use `iptables` rules installed at end hosts to selectively drop inbound packets that traversed specific links (according to DSCP markings). For example, we can configure an end host to drop 0.5% of all inbound packets that transited a particular core-to-Agg link. This has the effect of injecting faults at an arbitrary network location, yet impacting only the systems that we monitor.[2]

Our private testbed is a half-populated $k = 6$ fat tree, consisting of 3 (of 6) pods of 9 hosts each, connected via 9 core switches. Each core has three links, one to each pod. To inject faults, we use a 'bump in the wire' to perturb packets on a link. For example, consider a faulty core that drops (or delays) a random subset of packets traversing a link to an Agg. Thus, to inject a fault at that link, we replace the link connecting the core to the Agg with a link connecting it to a network bridge, which is in turn connected to the Agg. The bridge is a Linux host that forwards packets, randomly dropping (or delaying) a subset of packets in a given direction. We implement ECMP using source routing; the 5-tuple therefore allows us to determine the paths packets traverse in our testbed.

5.2 Motivating example

Over a five-minute interval in the Facebook datacenter, we induced faults on links from three different core switches to the instrumented Agg, denoted A, B and C. In particular, we induce faults in the order A, B, C, with a one-minute gap; we then removed the faults in reverse order with the same gap. Each fault is a random 0.5% packet drop (1 in 200 packets). In aggregate, this corresponds to an overall packet loss rate of $< 0.02\%$.

The top portion of Figure 3 depicts the t-test output for a single host. Flows are grouped according to incoming core downlink, and TCP retransmission statistics are aggregated into ten-second intervals. For a single host, for every non-faulty link (every series in black, and the faulty links before/after their respective faults) the t-test output is noisy, with p-values ranging from 0.15 to 1.0. However, during a fault event the p-value drops down to near 0. Close to 100% of the hosts flag the faulty links as guilty, with few false positives.

These guilty verdicts are sent to our controller. The controller runs the chi-squared test every 10 seconds using each core downlink as a category; it counts the number of *non-guilty* verdicts from the end hosts as metric and flags an error condition if the output p-value ≤ 0.05. Note that this flag is binary, indicating that there exists at least one active fault; the guilty verdict count must be consulted to identify the actual guilty link. The bottom portion of Figure 3 depicts the controller output using this mechanism. We depict the output of the test for all paths (in red), and for the set of paths excluding faulty paths $\{A\}$, $\{A, B\}$ and $\{A, B, C\}$ (in yellow, purple and black, respectively). These results indicate that the controller will flag the presence of a fault as long as there is at least one faulty link in the set of paths under consideration, thus supporting an iterative approach.

5.3 Speed and sensitivity

For our mechanism to be useful, it must be able to rapidly detect faults, even if the impact is slight. Moreover, ideally we could detect faults that result in increased latency, not just those that cause packet loss.

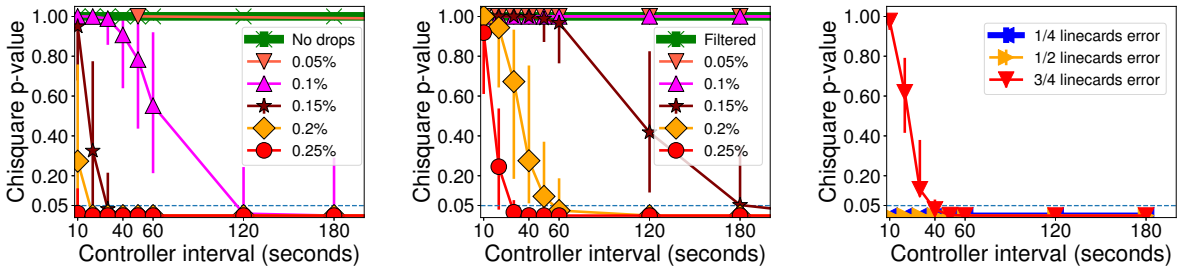
5.3.1 *Loss rate sensitivity*

We performed a sensitivity analysis on the amount of packet loss we can detect in the datacenter. While loss rates $\geq 0.5\%$ are caught by over 90% of hosts, we see a linear decrease in the number of guilty verdicts as the loss decreases past that point—at 0.1% drop rate, only $\approx 25\%$ of hosts detect a fault in any given 10-second interval, reducing our controller's effectiveness. However, we can account for this by prolonging the interval of the controller chi-square test. Figure 4a depicts the distribution of p-values outputted by the controller for a given loss rate and calculation interval. Each point with error bars depicts the median, p5 and p95 p-values outputted by the controller during a fault occurrence; each loss rate corresponds to a single series. We see that while a 0.25% loss is reliably caught with a 20-second interval, a 0.15% loss requires 40 seconds to be reliably captured; lower loss rates either take an unreasonably large (more than a minute) period of time to be caught or do not get detected at all. Note that in the no-fault case, no false positives are raised despite the increased monitoring interval.

5.3.2 *High latency detection*

In our private testbed, we induced delays for traffic traversing a particular core switch. To do this, we used Linux `tc-netem` on our 'bump-in-the-wire' network bridges to add constant delay varying from 100

[2]Note that while all monitored hosts will see the same loss *rate* across the link, the actual packets dropped may vary because `iptables` functions independently at each host.



(a) Single fault loss rate sensitivity. **(b)** Masked fault sensitivity. **(c)** Linecard faults.

Figure 4: Controller chi-square p-value convergence for various faults vs. controller interval length.

Request bytes	Latency msec	p50	p75	p95	p99
100	-	1643	1680	2369	2476
100	0.1	3197	3271	4745	4818
100	1.0	10400	10441	19077	19186
8000	-	4140	4778	619	7424
8000	0.1	6809	7510	9172	11754
8000	0.5	11720	14024	18367	21198

Table 3: `srtt_us` distribution vs. additional latency and request size. The no-additional-latency case is aggregated across all non-impacted links, while the others correspond to a single (faulty) link.

microseconds to 1 millisecond (a typical 4-MB switch buffer at 10 Gbps can incur a maximum packet latency of roughly 3 milliseconds before overflowing). We then ran client/server traffic with a single pod of 9 HHVM [5] servers serving static pages, and two pods (18 hosts) configured as web clients running Apache Benchmark [1]. Each server handled 180 simultaneous clients and served either small (100-B) or medium (8-KB) requests.

Table 3 depicts the distributions for TCP `srtt_us` TCP at a particular end host as a function of induced latency and request size. As in previous experiments, the distributions of non-impacted links are very close to each other, while the distribution of the latency heavy link is clearly differentiable. No drops were detected in the 100-B case; due to our use of 1-Gbps links, a handful of drops comparable to the no-fault case were detected in the 8-KB case. The modified KS-2 test operating over a 10-second interval correctly flags all intervals experiencing a latency increase, while avoiding false positives.

5.4 Precision and accuracy

Next, we demonstrate the precision and accuracy of the system in the presence of concurrent and correlated faults, as well as the absence of false positives.

5.4.1 Concurrent unequal faults

A large network will likely suffer concurrent faults of unequal magnitude, where the largest fault may mask the presence of others. While we surmise that the most visible fault will be easily diagnosable, ideally it would be possible to parallelize fault identification in this case.

We consider a scenario in the Facebook datacenter with two concurrent faults on distinct core-to-Agg switch links: one causing a packet loss rate of 0.5%, and one with a rate that varies from 0.25% to 0.15% (which we can easily identify in isolation). Using TCP retransmit statistics, the high-loss fault was flagged by almost all the hosts the entire time, while the flagging rate for the lower-impact fault depends roughly linearly on its magnitude. However, the drop-off in guilty verdicts is steeper in the presence of a masking, higher-impact fault. As a result, the 10- and 20-second controller intervals that flag the low-loss-rate faults in isolation no longer suffice.

Figure 4b depicts the controller chi-square p-value outputs for the set of paths *excluding* the one suffering from the readily identified larger fault; each series corresponds to a different loss rate for the smaller fault. The interval needed to detect such "masked" faults is longer; a 0.25% loss rate requires a 40-second interval to reliably be captured vs. 20 seconds in the unmasked case (Figure 4a), while a 0.15% rate requires over three minutes.

5.4.2 Large correlated faults

So far, we have considered faults impacting a small number of uncorrelated links. However, a single hardware fault can affect multiple links. For example, each Agg switch contains several core-facing linecards, each providing a subset of the switch's cross-pod capacity. A linecard-level fault would thus affect several uplinks. Similarly, a ToR-to-Agg switch link might be impacted, affecting a full quarter of the uplinks for that rack's hosts. Our approach relies on the student's t-test picking outliers from a given average, with the assumption that the average represents a non-broken link. However, certain faults might impact a vast swath of links, driving the average performance closer to that of the impacted links'.

To test this scenario, we induce linecard-level faults on 25%, 50%, 75% and 100% of the uplinks on the instrumented Agg switch in the Facebook datacenter. The per-link loss rate in each case was 0.25%. With 100% faulty links, our method finds no faults since no link is an outlier—a natural consequence of our approach. How-

Metric	Error	p50	p90	p95	p99
retx	-	0	0	1	2
retx	0.5%	1	3	4	5
cwnd	-	10	18	26	39
cwnd	0.5%	9	10	16	28

Table 4: TCP congestion window and retransmit distributions when binning by remote rack with a faulty rack inducing a 0.5% drop rate.

ever, in all other cases our approach works if the hosts declare paths faulty when the p-value ≤ 0.1. Figure 4c shows the controller performance for various linecard-level faults as a function of interval length. A 10-second interval captures the case where 25% of uplinks experience correlated issues, but intervals of 20 and 40 seconds, respectively are required in the 50% and 75% cases.

5.4.3 False positives

The longer our controller interval, the more sensitive we are to catching low-impact faults but the more likely we are to be subject to false positives. We ran our system in production in the absence of any (known) faults with intervals ranging from 10 seconds to an hour. Even with 30-minute intervals, the lowest p-value over 42 hours of data is 0.84; only one-hour intervals generated any false positives ($p \leq 0.05$) in our data. We note, however, that we need not support arbitrarily large intervals. Recall that an interval of roughly three minutes is enough to get at least an intermittent fault signal for a 0.1% loss rate.

5.4.4 Granular faults and alternative binnings

By default, our approach bins flow metrics by path. In certain cases, however, a fault may only impact a specific subset of traffic. For example, traffic from a particular subnet might exhibit microburst characteristics, periodically overflowing switch buffers and losing packets.

Alternative binnings can be employed to identify such "granular" faults. To illustrate, we induced a fault at a single cache server, in which packets from exactly one remote rack are dropped at a rate of 0.5%. We then binned traffic by remote rack. Table 4 depicts the distribution of congestion window and retransmit by remote rack; as before, the distributions for non-impacted bins are all close to each other. The KS-2 test and t-test successfully pick out the fault without false positives using the `cwnd` and retransmissions metrics respectively. Note such alternative binning can help diagnose faults even if there is no path diversity—in this case, the alternatives are provided by application load balancing.

5.5 Small-scale deployment experience

While our experiments focus on injected failures, we are obviously interested in determining whether our system can successfully detect and localize network anomalies "in the wild". Thus, we examine the performance of our system over a relatively long time period (in the absence of any induced failures) to answer the following questions: "Does our system detect performance anomalies?" "Do non-issues trigger false positives?" "Do we notice anomalies before or after Facebook's existing fault-detection services catch it?"

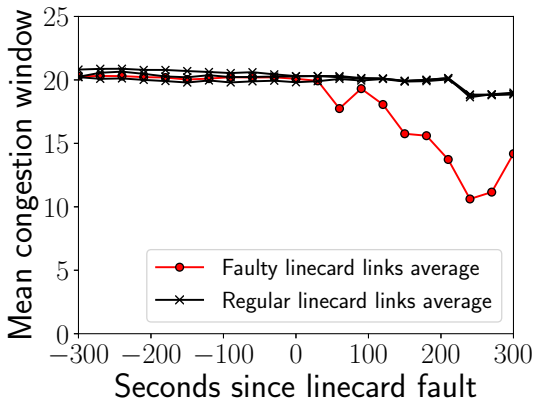


Figure 5: Mean `cwnd` per (host,link) during linecard fault.

To answer these questions, we deployed our system on 30 hosts for a two-week period in early 2017. As an experimental system, deployment was necessarily restricted; our limited detection surface thus impacted our chance of detecting partial faults. An operator of another large-scale datacenter suggests that, in their experience, partial failures occur at a rate of roughly 10 per day in a network containing O(1M) hosts [28]. It is not surprising, then, that our two-week trial on only 30 hosts did not uncover any partial faults. We were, however, able to derive useful operational experience that we relate below.

5.5.1 Response to organic failure

On January 25th, 2017, the software agent managing a switch linecard that our system was monitoring failed. The failure had no immediate impact on traffic; the data-plane continued to forward traffic according to the most recent ruleset installed by the agent. Thus, the initial failure is invisible to and explicitly outside the scope of our tool, which focuses only on faults that impact traffic.

Roughly a minute later, however, as the BGP peerings between the linecard and its neighbors began to time out, traffic was preemptively routed away from the impacted linecard. Thus, applications saw no disruption in service despite the unresponsive linecard control plane. Yet, we observe that as traffic was routed away from the failed linecard, the distributions of TCP's `cwnd` and `ssthresh` metrics for the traffic remaining on the linecard's links rapidly diverged from the values on other links in the equivalence set. Figure 5 depicts the per-link mean congestion window measured by every host, aggregated per-linecard and averaged across every corresponding host, with the afflicted linecard colored red.

The deviations are immediate and significant, with the mean `cwnd` dropping over 10% in the first interval after

the majority of traffic is routed away, and continually diverging from the working links thereafter. Furthermore, the volume of measured flows at each host traversing the afflicted linecard rapidly drops from O(1000s) to O(10s) per link. By contrast, one of Facebook's monitoring systems, NetNORAD [6], took several minutes to detect the unresponsive linecard control plane and raise an alert.

It is important to note that in this case, we did not catch the underlying software fault ourselves; that honor goes to BGP timeouts. However, we do observe a sudden shift in TCP statistics in real time as traffic is routed away, as our system was designed to do. With respect to our stated goal—to pinpoint the links responsible for deleterious traffic impact—our system performs as expected. Thus, this anecdote shows that our system can compliment existing fault-detection systems, and provide rapid notification of significant changes in network conditions on a per-link or per-device basis.

5.5.2 Filtering normal congestion events

During the monitoring period, no other faults were detected by the system. While a small number of false positives were generated in every interval, the controller filters out these indications since the noise is spread across the monitored links. However, we noticed that the number of false positives had periodic local maxima around 1200 and 1700 GMT. Furthermore, these were correlated with the raw (i.e., independent of flow/path) TCP retransmit counts tracked by the end hosts. Given that they occurred at similar times each day and were evenly spread across all the monitored hosts, we surmise that these retransmits were not due to network faults, but organically occurring congestion. This experience provides some confidence that our system effectively distinguishes between transient congestion and partial faults.

6. APPLICABILITY

Here, we consider the applicability of our approach in various challenging scenarios, e.g., datacenters with heterogeneous traffic patterns, topologies less amenable to single-marking path discovery (either by design or due to failed links re-routing traffic), virtualized multi-tenant environments, and more. We first list some conditions to which our approach is resilient. Subsequently, we clarify the extent to which traffic homogeneity, link failures and topology impact the suitability of our approach. We conclude with a discussion on known limitations.

6.1 Surmountable issues

While we have access to only one production environment, we have conducted sensitivity analyses in our testbed to consider alternative deployments. Due to space limits, we summarize our findings here, but provide more extensive discussion in Appendix B.

1. **VMs and high CPU utilization**: Our datacenter tests were on bare metal; we find the approach works equally well in a hypervisor-based environment, even when the hardware CPU is fully taxed.
2. **Mixed, over-subscribed and uneven traffic**: The Facebook datacenter lacks saturated links and uneven load. We consider a mixed workload with latency sensitive and core-saturating bulk traffic, where server load ranges from $1\times$—$16\times$.
3. **TCP settings**: Datacenter hosts employ NIC offload features such as TSO. Turning these optimizations off does not obscure TCP or timing-based signals; neither does varying buffer sizes across three orders of magnitude (16 MB—16 KB).

6.2 Traffic homogeneity

Facebook traffic is highly load balanced [35], aiding our outlier-based detection approach. We are optimistic, however, that our method is also applicable to datacenters with more heterogeneous and variable workloads. That said, outlier analysis is unlikely to succeed in the presence of heterogeneous traffic if we do not carefully pick the links or switches that we compare against each other—specifically, if we fail to form a valid ES.

In the case of our experiments at Facebook, the ES we used was the set of downlinks from the network core into a pod. ECMP routing ensured that these links did form an ES; all cross-pod flows had an equal chance of mapping to any of these links, and the path from these links down to the hosts were equal cost. This characteristic notably holds true regardless of the specific mix of traffic present. Thus, we hypothesize that on any network where such an ES can be formed, our approach works regardless of traffic homogeneity. To demonstrate this, we ran our fat-tree testbed with a worst case scenario of heterogeneous traffic: running synthetic bulk transfer and latency sensitive RPC traffic, with heavy traffic skew (with per host-load ranging from 1–16$\times$ the minimum load). Furthermore, we overloaded the network core by artificially reducing the number of links. Even in this case, our t-test classifier operating on the `select()` latency metric was able to successfully differentiate the outlier link.

6.3 Link failures

In addition to being able to detect and localize partial faults, our system must be able to account for total link failures, which can confound our ability to determine a flow's path through the network due to re-routing. Consider the outcome of a total link failure on the fate of traffic routed via that link. There are three possible outcomes for such traffic: (1) traffic is redirected at a prior hop to a working alternative path, (2) traffic is re-routed by the switch containing the dead-end link to a backup non-shortest path, and (3) traffic is black holed and the flow stops (the application layer might restart the flow).

Cases (1) and (3) do not affect our approach. ECMP routing will ensure that flows are evenly distributed among the surviving links, which still form an ES (al-

beit one smaller in size than before the failure). Case (2) can impact our approach in two ways. First, traffic taking a longer path will likely see worse performance compared to the rest of the traffic that traverses links on the backup path—harming the average performance on that path. Moreover, backup path performance might drop due to unfair loading as more flows join. Presuming rerouting is not silent (e.g., because it is effected by BGP), the former effect can be accounted for; traffic using backup routes can be marked by switches and ignored in the end-host t-test computation. The latter can be mitigated by careful design: rather than loading a single backup path unfairly, the load can be evenly distributed in the rest of the pod. Even if an imbalance cannot be avoided, two smaller ESes can yet be formed: one with links handling rerouted traffic, and one without.

6.4 Topologies

Our system leverages the details of Facebook's datacenter topology to obtain full path info with a single marking identifying the transited core switch. The topology also allows us to form equivalence sets for each pod by considering the core-to-pod downlinks. Other networks might provide more challenging environments (e.g. middleboxes or software load balancers [31] might redirect some traffic; different pods might have varying internal layouts; links might fail) that confound the ability to form equivalence sets. In an extreme case, a Jellyfish-like topology [38] might make it extremely difficult to both extract path information and form ESes.

In certain cases, though, even networks with unruly layouts and routing can be analyzed by our approach. Consider a hybrid network consisting of a Jellyfish-like subset. For example, suppose a single switch in the Jellyfish sub-network is connected to every core switch in a multi-rooted tree, with identical link bandwidths. While we cannot reason about traffic internal to the Jellyfish, we can still form an ES for the links from the single switch connecting to the regular topology, for all traffic flowing into the regular sub-network. No matter how chaotic the situation inside the Jellyfish network, the traffic should be evenly distributed across core switches in the regular sub-network, and from then on the paths are equivalent.

Note that here, we only consider the subset of the path that lies within the regular topology. As long as an ES can be formed, the path behind it can be considered as a black box. Thus, we argue that even on topologies where we can't find the full path, or where inequalities in path cost exist, we can run our approach on subsets of the topology where our requirements do hold.

6.5 Limitations

On the other hand, there are certain cases where our current approach falls short. To begin, we presume we are able to collect end-host metrics that reflect network performance. While we believe our current metrics cover the vast majority of existing deployments, we have not yet explored RDMA-based applications or datacenter fabrics. Our current metrics also have limits to their precision; while we can detect 0.1% drop rates in the datacenter we studied, past a certain point we are unable to discern faults from noise. Moreover, the production datacenter is well provisioned, so fault-free performance is stable, even in the tail. Hence, we do not consider the inability to detect minor impairments to be a critical flaw: If the impacts of a fault are statistically indistinguishable from background network behavior, then the severity of the error might not be critical enough to warrant immediate response. Datacenters operating closer to capacity, however, may both exhibit less stable fault-free behavior, as well as require greater fault-detection sensitivity.

Despite extensive use of ECMP and application load-balancing techniques, datacenter networks with mixed workloads may include links that see more traffic and congestion than others. That said, we have not encountered enough variability in load in the network core and aggregation layers to trigger enough false positives to confound the accuracy of our method in either production or under our testbed, and thus cannot yet quantify under what circumstances this degradation in detection performance would occur. Furthermore, for links lacking alternatives—such as an end host to top-of-rack access link in the absence of multihoming—we cannot pick out an outlier by definition since there is only one link to analyze. We can, however, still perform our analysis on different groupings within traffic for that link; for example, when traffic to a particular subnet is impacted.

7. SUMMARY

We have demonstrated a method to use the regularity inherent in a real-world deployment to help rapidly detect and localize the effects of failures in a production datacenter network. In particular, we have shown that we can do it with reasonable computational overhead and ease of deployment. We believe improvements in network core visibility, together with coordinating end-host performance monitoring with centralized network control, can aid the task of managing network reliability.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation through grants CNS-1422240 and CNS-1564185. We are indebted to Aaron Schulman, Ariana Mirian, Danny Huang, Sunjay Cauligi, our shepherd, Dave Maltz, and the anonymous reviewers for their feedback. Alex Eckert, Alexei Starovoitov, Daniel Neiter, Hany Morsy, Jimmy Williams, Nick Davies, Petr Lapukhov and Yuliy Pisetsky provided invaluable insight into the inner workings of Facebook services. Finally, we thank Omar Baldonado for his continuing support.

8. REFERENCES

[1] ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[2] BPF Compiler Collection (BCC). https://github.com/iovisor/bcc/.

[3] Extending extended BPF. https://lwn.net/Articles/603983/.

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

[5] HHVM. http://hhvm.com.

[6] A. Adams, P. Lapukhov, and H. Zeng. https://code.facebook.com/posts/1534350660228025/netnorad-troubleshooting-networks-via-end-to-end-probing/.

[7] M. Aguilera, J. Mogul, J. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. SOSP, Bolton Landing, NY, USA, 2003. ACM.

[8] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. SIGCOMM, Seattle, WA, USA, 2008. ACM.

[9] A. Andreyev. Introducing data center fabric, the next-generation Facebook data center network. https://code.facebook.com/posts/360346274145943, 2014.

[10] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred. Taking the Blame Game out of Data Centers Operations with NetPoirot. SIGCOMM, Florianopolis, Brazil, 2016. ACM.

[11] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards Highly Reliable Enterprise Network Services via Inference of Multi-level Dependencies. SIGCOMM, Kyoto, Japan, 2007. ACM.

[12] D. Banerjee, V. Madduri, and M. Srivatsa. A Framework for Distributed Monitoring and Root Cause Analysis for Large IP Networks. SRDS, Niagara Falls, NY, USA, 2009. IEEE Computer Society.

[13] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based Faliure and Evolution Management. NSDI, San Francisco, California, 2004. USENIX Association.

[14] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. DSN, Bethesda, MD, USA, 2002. IEEE Computer Society.

[15] Cisco. BGP Support for TTL Security Check. http://www.cisco.com/c/en/us/td/docs/ios/12_2s/feature/guide/fs_btsh.html.

[16] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI, San Francisco, CA, 2004. USENIX Association.

[17] N. Duffield, F. L. Presti, V. Paxson, and D. Towsley. Network Loss Tomography Using Striped Unicast Probes. *ACM Transactions on Networking*, 14(4), Aug. 2006.

[18] G. Forman, M. Jain, M. Mansouri-Samani, J. Martinka, and A. C. Snoeren. Automated Whole-System Diagnosis of Distributed Services Using Model-Based Reasoning. DSOM, Newark, DE, Oct. 1998.

[19] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. SIGCOMM, London, United Kingdom, 2015. ACM.

[20] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. NSDI, Seattle, WA, 2014. USENIX Association.

[21] T. Hoff. Latency Is Everywhere And It Costs You Sales - How To Crush It. http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it.

[22] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. EuroSys, Lisbon, Portugal, 2007. ACM.

[23] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene. FlowBender: Flow-level Adaptive Routing for Improved Latency and Throughput in Datacenter Networks. CoNEXT, Sydney, Australia, 2014. ACM.

[24] S. Kandula, D. Katabi, and J.-P. Vasseur. Shrink: A Tool for Failure Diagnosis in IP Networks. MineNet, Philadelphia, Pennsylvania, USA, 2005. ACM.

[25] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. IP Fault Localization via Risk Modeling. NSDI, Boston, MA, USA, 2005. USENIX Association.

[26] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Detection and Localization of Network Black Holes. INFOCOM, Anchorage, AK, USA, 2007. IEEE Computer Society.

[27] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Fault Localization via Risk Modeling. *IEEE Trans. Dependable Secur. Comput.*, 7(4), Oct. 2010.

[28] D. Maltz. Private communication, Feb. 2017.

[29] V. Mann, A. Vishnoi, and S. Bidkar. Living on the edge: Monitoring network flows at the edge in cloud data centers. COMSNETS, Bangalore, India, 2013. IEEE.

[30] R. N. Mysore, R. Mahajan, A. Vahdat, and G. Varghese. Gestalt: Fast, Unified Fault Localization for Networked Systems. USENIX ATC, Philadelphia, PA, 2014. USENIX Association.

[31] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud Scale Load Balancing. SIGCOMM, Hong Kong, China, 2013. ACM.

[32] S. Radhakrishnan, M. Tewari, R. Kapoor, G. Porter, and A. Vahdat. Dahu: Commodity Switches for Direct Connect Data Center Networks. ANCS, San Jose, California, USA, 2013. IEEE Press.

[33] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. WAP5: Black-box Performance Debugging for Wide-area Systems. WWW, Edinburgh, Scotland, 2006. ACM.

[34] M. Roughan, T. Griffin, Z. M. Mao, A. Greenberg, and B. Freeman. IP Forwarding Anomalies and Improving Their Detection Using Multiple Data Sources. NetT, Portland, Oregon, USA, 2004. ACM.

[35] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the Social Network's (Datacenter) Network. SIGCOMM, London, United Kingdom, 2015. ACM.

[36] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. SIGCOMM, Stockholm, Sweden, 2000. ACM.

[37] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. SIGCOMM, London, United Kingdom, 2015. ACM.

[38] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking Data Centers Randomly. NSDI, San Jose, CA, 2012. USENIX Association.

[39] D. Turner, K. Levchenko, J. C. Mogul, S. Savage, and A. C. Snoeren. On Failure in Managed Enterprise Networks. Technical report, Hewlett-Packard Labs, May 2012.

[40] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage. California Fault Lines: Understanding the Causes and Impact of Network Failures. SIGCOMM, New Delhi, India, 2010. ACM.

[41] B. P. Welford. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics*, 4(3), 1962.

[42] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang. NetPilot: Automating Datacenter Network Failure Mitigation. SIGCOMM, Helsinki, Finland, 2012. ACM.

[43] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. SIGCOMM, Helsinki, Finland, 2012. ACM.

[44] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-Level Telemetry in Large Datacenter Networks. SIGCOMM, London, United Kingdom, 2015. ACM.

A. ALTERNATIVE METHODS

Alternative methods abound for investigating datacenter faults. One appealing option is to couple switch counters with programmable switches. For example, suppose a switch is configured such that when a packet is dropped, either due to error or queuing pressure, it is probabilistically sampled to the switch CPU (note that with switches shifting millions of packets a second, examining all packets leads to unacceptable CPU overhead). Additionally, some switches can be configured to sample packets from high occupancy queues. Thus, counters could indicate a fault, while sampled packets could be used to build a picture of which traffic is impacted.

While this method can alert network operators to faulty links, it is slow in determining which traffic is impacted. Suppose a link carrying 2,000,000 packets per second develops a 0.5% drop rate, leading to 10,000 drops per second. Such a link might be carrying tens of thousands of flows, however. With a relatively high 1 in 100 sampling rate (and thus, 100 sampled packets per second), and with just 10,000 flows carried (a large underestimation) it would take around 100 seconds to determine all the flows if the sampling was perfect and captured a different 5-tuple each time. Furthermore, this would be subject to sampling bias; a lightweight flow carrying control traffic might lose packets but fly under the radar of sampled dropped packets. One heavy handed potential approach could be to disable the entire link; however, consider the case of a partial fault only affecting a subset of traffic (for example, a misconfigured routing rule). In such a case, disabling the entire link would penalize all traffic routed through that link without providing any insight to the underlying problem.

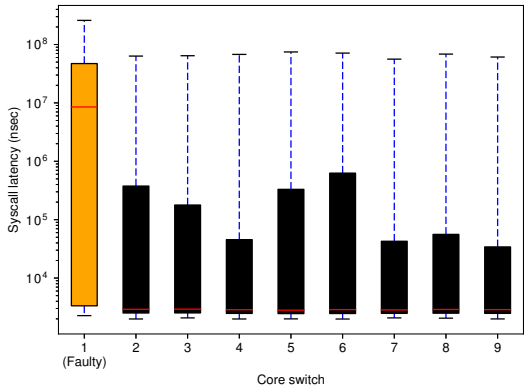


Figure 6: `select()` latency distribution per core switch for the two-VM CPU stress test.

For contrast, our proposed system can quickly identify all the flows impacted by a link error in a less than a minute in several tested cases. In the case of granular errors affecting only a subset of traffic (by source or destination subnet, application port, queuing policy, etc.) our mechanism can still detect outliers, enabling end hosts to reroute around damaged links for afflicted traffic as a stopgap measure. Note that switch counters can be used in conjunction with our methods; features of sampled dropped or latency enduring packets could be used to guide our system's binning of flows in order to converge on which traffic is affected by link faults even quicker. Furthermore, our proposed system is ASIC agnostic since we do not rely on the features of any given chipset. Finally, it is robust to unreliable reporting by switches, as well as the uncertainty of counters that might arise within environments using cut-through routing.

B. METRIC ROBUSTNESS

Here, we demonstrate the effectiveness of our approach in the presence of pathological confounding factors. These experiments are performed on our private testbed, since we could not induce harmful configuration changes in production traffic. We focus on syscall latency metrics, which are less obviously robust to many of thse factors. To save space, we omit similarly strong results using TCP statistics.

B.1 CPU utilization

Datacenter networks run a variety of applications, frequently with stringent CPU requirements and high utilization. To be general, our approach needs to cope with high utilization figures. Furthermore, while some datacenters run applications on bare metal hardware, a significant number of installations use virtual machines.

In order to ascertain the impact of virtual machine hardware and high cpu utilization on our approach, we set up an experiment where each host in our private testbed runs two virtual machines, `cpuvm` and `netvm`. Each instance of `cpuvm` runs a variety of simultaneous CPU intensive tasks: specifically, a fully loaded `mysql` server instance (with multiple local clients), a scheduler-intensive multithreading benchmark, and an ALU-operation-intensive math benchmark. This causes the CPU utilization of the bare metal host to go to 100%. Meanwhile, each instance of `netvm` runs a bidirectional client-server communication pattern using flow size distributions prevalent in certain datacenters [35]. This situation is analogous to the case where one VM is shuffling data for a map-reduce job, while another is actively processing information, for example. Figure 6 depicts the distribution of `select()` latencies across each core switch in the presence of this extreme CPU stress. The distribution corresponding with paths through the faulty switch are clearly distinguishable, with a median value over three orders of magnitude greater than the no-error case. This corresponds to a 100% accuracy rate of flagging the faulty link using our chi-squared test over 1-minute intervals for a 5-minute long test.

B.2 Oversubscription and uneven load

Thus far, all our experiments have networks without saturated core links. While this is a desirable quality in real deployments, it might not be feasible depending on the amount of oversubscription built into the network. Furthermore, in heterogeneous environments such as multi-tenant datacenters, it is conceivable that not every end-host is participating in monitoring traffic metrics. Finally, not every system that is being monitored is subject the same load.

To determine how our approach deals with these challenges, we devised an experiment in our private testbed where 1/3rd of our 27 hosts are no longer participating in monitoring. Instead, these hosts suffuse the network with a large amount of background traffic, with each so-called 'background host' communicating with all other background hosts. Each background host sends a large amount of bulk traffic; either rate limited to a fraction of access link capacity, or unbound and limited only by link capacity. In conjunction with this, our network core is reduced to 1/3rd of normal capacity: from 9 core switches to 3. Thus, the background hosts can, by themselves, saturate core capacity. Our remaining 18 servers continue to run the bidirectional client-server network pattern, with one important difference: rather than all servers having the same amount of load, they are partitioned into three classes. The first class chooses flow sizes from the Facebook cache server flow size distribution [35]; the other two classes choose flow sizes from larger $4\times$ and $16\times$ multiples of the Facebook distribution. In other words, the $4\times$ distribution is simply the normal distribution, except every value for flow size being multiplied by 4.

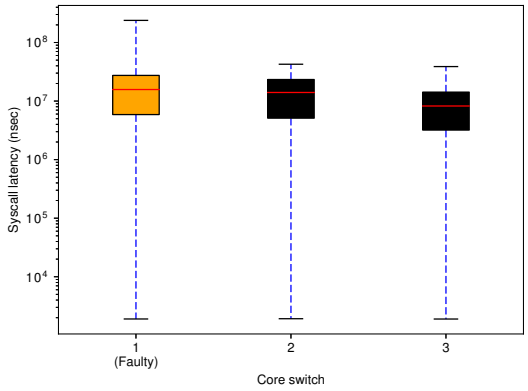


Figure 7: `select()` latency distribution per core switch for the oversubscribed background traffic uneven server test.

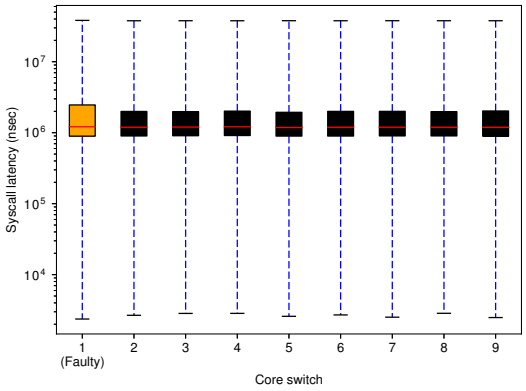


Figure 8: `select()` latency distribution per core switch for the 10-Gbps 16-KB socket send buffer test.

Thus, the link utilizations of the $4\times$ and $16\times$ distributions are correspondingly higher than normal. While the normal distribution has a link utilization of roughly 100 Mbps in this case, the $4\times$ case and $16\times$ cases bring us to roughly 40% and 100% link utilization respectively.

Figure 7 depicts the `select()` latency distribution for this scenario as a multi-series CDF. Each series represents the distribution for a single network path; there is one faulty path and two working paths in this case. The x-axis depicts latency in nanoseconds. Despite the combination of confounding factors, the distribution for the faulty path remains significantly higher (shifted to the right in the figure) than the non-faulty paths (which possess roughly equivalent distributions). Correspondingly, the signal derived from the select() latencies remains useful. Using 1-minute intervals, we note that the chi-squared test hovered at close to 0 when we considered all paths, and close to 1 when the faulty path was removed from the set of candidate paths. This indicates that we remain successful at finding the faulty link even under these circumstances.

B.3 Sensitivity to TCP send buffer size

Our system call latency approach uses the latency of `select()`/`epoll()` calls by hosts that are sending data over the network as a signal. As a reminder, when an application calls `select()`/`epoll()` (or `send()`) on a socket, the call returns (or successfully sends data) when there is enough room in the per-flow socket buffer to store the data being sent. Thus, the distribution of results is fundamentally impacted by the size of the per-flow socket buffer. Note that with the advent of send-side autotuning in modern operating systems, the size of this buffer is not static. In Linux, the kernel is configured with a minimum size, a default (initial) size and a maximum size, with the size growing and shrinking as necessary.

By default, our private testbed systems are configured with a maximum send buffer size of 4 MB. Values for the median buffer size vary from 1.5–2.5 MB for our server hosts and 0.8–1.2 MB for our client hosts in our bidirectional traffic pattern. To determine the sensitivity of our approach to buffer size, we first increased the buffer size to values of 8 and 16 MB. Then, we decreased the buffer size significantly, to values of 52, 26 and 16 KB (note that 16 KB is the default initial size). We noticed that TCP autotune prevents the buffer from growing too large even if the maximum is set significantly higher than default; accordingly, raising the maximum had little to no impact.

Reducing the maximum buffer size, however, is expected to have a larger impact on our metrics. Intuitively, a smaller buffer should correspond with higher overall buffer occupancy; thus, a larger percentage of `select()` and `send()` calls would need to wait until the buffer is drained. This should have the two-fold effect of shifting the `select()` latency distributions to the right—since more calls to `select()` wait for longer—and decreasing the distance between a faulty-path distribution and the normal case.

To test this out, we ran our bidirectional client-server traffic pattern using 1-Gbps NICs on our $k = 6$ testbed, and using 10-Gbps NICs on our three system setup. We ran a low-intensity test (roughly 50 and 300 Mbps link utilization per client and server host, respectively) and a high-intensity test (roughly 1–2 and 5–7 Gbps for client and server hosts respectively in the 10-Gbps case, and saturated links for servers in the 1-Gbps case). We noted that in all the low-intensity test cases, and in the 1-Gbps high-intensity cases, we were still able to derive a clear signal and successfully pick out the faulty link using 1-minute intervals. In our worst-case scenario of 10-Gbps links and a 16-KB send buffer size, we note a significant

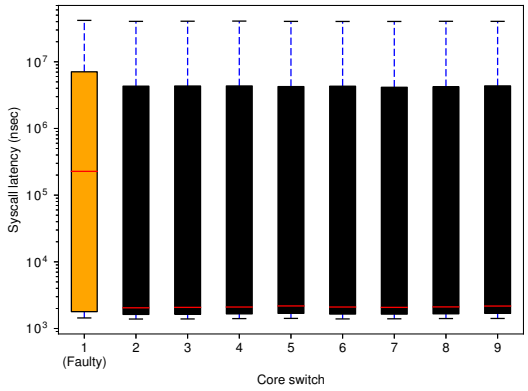


Figure 9: `select()` latency distribution per core switch for the no-offload test with CPU stress.

shifting of the distribution to the right by roughly three orders of magnitude, as predicted, as depicted in Figure 8. While the faulty-path distribution is closer to the non-faulty distributions, there is still enough divergence to clearly differentiate it from the pack, albeit requiring our statistical tests to run over longer time intervals. We note, however, that such small send socket buffer sizes are unlikely in real-world deployments.

B.4 The effect of NIC offloads

Contemporary NICs have a variety of processing offload features designed to cut down on end host CPU utilization (which we've already discussed can be heavily utilized in a datacenter environment). For example, TCP discretizes a continuous stream into individual packets determined by end to end path MTU; a feature called "TCP Segmentation Offload" allows the NIC to perform this discretization, saving the CPU the effort.

NICs typically implement three common offload features: TCP Segmentation Offload, Generic Segmentation Offload and Generic Receive Offload. All our previous experiments had these features turned on in every end host. To test the sensitivity of our approach to the availability and state of these features, we re-ran the bidirectional client-server traffic pattern with all three offload features turned off. Figure 9 depicts the `select()` latency distribution for two client hosts and one server host within our private testbed with all offloads turned off and a CPU utilization of 100% using the combination of programs described in the CPU stress test earlier. This scenario represents a sort of worst case for our approach due to the high CPU utilization required to push multiple gigabits of traffic without offload; despite this, our approach is still able to correctly flag the faulty path over a 30-second interval.

C. GENERAL-CASE TOPOLOGIES

The characteristics of the production datacenter we examined in this work lends itself to relatively simple diagnosis of which link is faulty based on per-link performance metric binning; however, alternative scenarios may exist where either traffic is not as evenly load balanced, or a different topology complicates the ability to subdivide link by hierarchy for the purposes of comparison (for example, different pods in a network might have different tree depths).

Our method fundamentally depends on finding roughly equivalent binnings for flow metrics, such that outliers can be found. In the case of a network with hotspots or complicated topologies, which can confound this process, we can still make headway by considering links at every multipath decision point as a single equivalency group. If one of these links exhibits outlier performance, the flows traversing the link can be marked individually as faulty rather than the link as a whole, and the flow and the traversed path can be submitted as input to a graph-based fault localization algorithm such as SCORE [25] or Gestalt [30]. We leave the evaluation of such a scenario to future work.

Clipper: A Low-Latency Online Prediction Serving System

Daniel Crankshaw*, Xin Wang*, Giulio Zhou*
Michael J. Franklin*†, Joseph E. Gonzalez*, Ion Stoica*
*UC Berkeley †The University of Chicago

Abstract

Machine learning is being deployed in a growing number of applications which demand real-time, accurate, and robust predictions under heavy query load. However, most machine learning frameworks and systems only address model training and not deployment.

In this paper, we introduce Clipper, a general-purpose low-latency prediction serving system. Interposing between end-user applications and a wide range of machine learning frameworks, Clipper introduces a modular architecture to simplify model deployment across frameworks and applications. Furthermore, by introducing caching, batching, and adaptive model selection techniques, Clipper reduces prediction latency and improves prediction throughput, accuracy, and robustness without modifying the underlying machine learning frameworks. We evaluate Clipper on four common machine learning benchmark datasets and demonstrate its ability to meet the latency, accuracy, and throughput demands of online serving applications. Finally, we compare Clipper to the Tensorflow Serving system and demonstrate that we are able to achieve comparable throughput and latency while enabling model composition and online learning to improve accuracy and render more robust predictions.

1 Introduction

The past few years have seen an explosion of applications driven by machine learning, including recommendation systems [28, 60], voice assistants [18, 26, 55], and ad-targeting [3, 27]. These applications depend on two stages of machine learning: *training* and *inference*. Training is the process of building a model from data (e.g., movie ratings). Inference is the process of using the model to make a prediction given an input (e.g., predict a user's rating for a movie). While training is often computationally expensive, requiring multiple passes over potentially large datasets, inference is often assumed to be inexpensive. Conversely, while it is acceptable for training to take hours to days to complete, inference must run in real-time, often on orders of magnitude more queries than during training, and is typically part of user-facing applications.

For example, consider an online news organization that wants to deploy a content recommendation service to personalize the presentation of content. Ideally, the service should be able to recommend articles at interactive latencies (<100ms) [64], scale to large and growing user populations, sustain the throughput demands of flash crowds driven by breaking news, and provide accurate predictions as the news cycle and reader interests evolve.

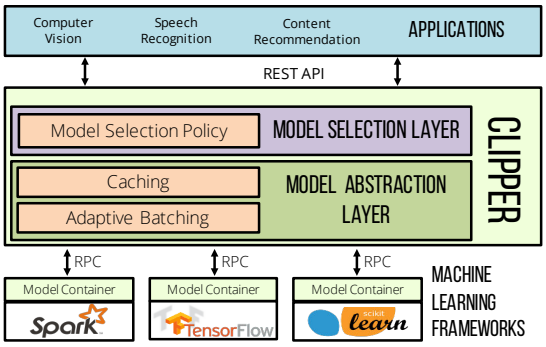


Figure 1: The Clipper Architecture.

The challenges of developing these services differ between the training and inference stages. On the training side, developers must choose from a bewildering array of machine learning frameworks with diverse APIs, models, algorithms, and hardware requirements. Furthermore, they may often need to migrate between models and frameworks as new, more accurate techniques are developed. Once trained, models must be *deployed* to a prediction serving system to provide low-latency predictions at scale.

Unlike model development, which is supported by sophisticated infrastructure, theory, and systems, model deployment and prediction-serving have received relatively little attention. Developers must cobble together the necessary pieces from various systems components, and must integrate and support inference across multiple, evolving frameworks, all while coping with ever-increasing demands for scalability and responsiveness. As a result, the deployment, optimization, and maintenance of machine learning services is difficult and error-prone.

To address these challenges, we propose Clipper, a *layered architecture* system (Figure 1) that reduces the complexity of implementing a prediction serving stack and achieves three crucial properties of a prediction serving system: *low latencies*, *high throughputs*, and *improved accuracy*. Clipper is divided into two layers: (1) the model abstraction layer, and (2) the model selection layer. The first layer exposes a common API that abstracts away the heterogeneity of existing ML frameworks and models.

Consequently, models can be modified or swapped transparently to the application. The model selection layer sits above the model abstraction layer and dynamically selects and combines predictions across competing models to provide more accurate and robust predictions.

To achieve low latency, high throughput predictions, Clipper implements a range of optimizations. In the model abstraction layer, Clipper caches predictions on a per-model basis and implements adaptive batching to maximize throughput given a query latency target. In the model selection layer, Clipper implements techniques to improve prediction accuracy and latency. To improve accuracy, Clipper exploits bandit and ensemble methods to robustly select and combine predictions from multiple models and estimate prediction uncertainty. In addition, Clipper is able to adapt the model selection independently for each user or session. To improve latency, the model selection layer adopts a straggler mitigation technique to render predictions without waiting for slow models. Because of this layered design, neither the end-user applications nor the underlying machine learning frameworks need to be modified to take advantage of these optimizations.

We implemented Clipper in Rust and added support for several of the most widely used machine learning frameworks: Apache Spark MLLib [40], Scikit-Learn [51], Caffe [31], TensorFlow [1], and HTK [63]. While these frameworks span multiple application domains, programming languages, and system requirements, each was added using fewer than 25 lines of code.

We evaluate Clipper using four common machine learning benchmark datasets and demonstrate that Clipper is able to render low and bounded latency predictions (<20ms), scale to many deployed models even across machines, quickly select and adapt the best combination of models, and dynamically trade-off accuracy and latency under heavy query load. We compare Clipper to the Google TensorFlow Serving system [59], an industrial grade prediction serving system tightly integrated with the TensorFlow training framework. We demonstrate that Clipper's modular design and broad functionality impose minimal performance cost, achieving comparable prediction throughput and latency to TensorFlow Serving while supporting substantially more functionality. In summary, our key contributions are:

- A layered architecture that abstracts away the complexity associated with serving predictions in existing machine learning frameworks (§3).
- A set of novel techniques to reduce and bound latency while maximizing throughput that generalize across machine learning frameworks (§4).
- A model selection layer that enables online model selection and composition to provide robust and accurate predictions for interactive applications (§5).

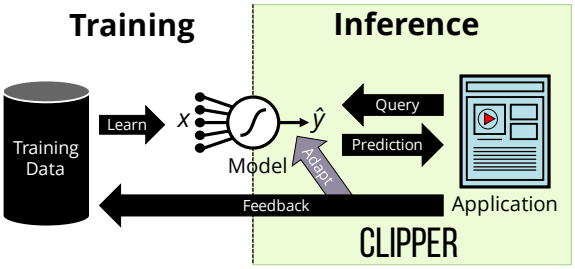


Figure 2: Machine Learning Lifecycle.

2 Applications and Challenges

The machine learning life-cycle (Figure 2) can be divided into two distinct phases: *training* and *inference*. Training is the process of estimating a model from data. Training is often computationally expensive requiring multiple passes over large datasets and can take hours or even days to complete [11, 29, 41]. Much of the innovation in systems for machine learning has focused on model training with the development of systems like Apache Spark [65], the Parameter Server [38], PowerGraph [25], and Adam [14].

A wide range of machine learning frameworks have been developed to address the challenges of training. Many specialize in particular models such as TensorFlow [1] for deep learning or Vowpal Wabbit [34] for large linear models. Others are specialized for specific application domains such as Caffe [31] for computer vision or HTK [63] for speech recognition. Typically, these frameworks leverage advances in parallel and distributed systems to scale the training process.

Inference is the process of evaluating a model to render predictions. In contrast to training, inference does not involve complex iterative algorithms and is therefore generally assumed to be easy. As a consequence, there is little research studying the process of inference and most machine learning frameworks provide only basic support for offline batch inference – often with the singular goal of evaluating the model training algorithm. However, scalable, accurate, and reliable inference presents fundamental system challenges that will likely dominate the challenges of training as machine learning adoption increases. In this paper we focus on the less studied but increasingly important challenges of *inference*.

2.1 Application Workloads

To illustrate the challenges of inference and provide a benchmark on which to evaluate Clipper, we describe two canonical real-world applications of machine learning: *object recognition* and *speech recognition*.

Object Recognition

Advances in deep learning have spurred rapid progress in computer vision, especially in object recognition prob-

lems – the task of identifying and labeling the objects in a picture. Object recognition models form an important building block in many computer vision applications ranging from image search to self-driving cars.

As users interact with these applications, they provide feedback about the accuracy of the predictions, either by explicitly labeling images (e.g., tagging a user in an image) or implicitly by indicating whether the provided prediction was correct (e.g., clicking on a suggested image in a search). Incorporating this feedback quickly can be essential to eliminating failing models and providing a more personalized experience for users.

Benchmark Applications: We use the well studied MNIST [35], CIFAR-10 [32], and ImageNet [49] datasets to evaluate increasingly difficult object recognition tasks with correspondingly larger inputs. For each dataset, the prediction task requires identifying the correct label for an image based on its pixel values. MNIST is a common baseline dataset used to demonstrate the potential of a new algorithm or technique, and both deep learning and more classical machine learning models perform well on MNIST. On the other hand, for CIFAR-10 and Imagenet, deep learning significantly outperforms other methods. By using three different datasets, we evaluate Clipper's performance when serving models that have a wide variety of computational requirements and accuracies.

Automatic Speech Recognition

Another successful application of machine learning is automatic speech recognition. A speech recognition model is a function from a spoken audio signal to the corresponding sequence of words. Speech recognition models can be relatively large [10] and are often composed of many complex sub-models trained using specialized speech recognition frameworks (e.g., HTK [63]). Speech recognition models are also often personalized to individual users to accommodate variations in dialect and accent.

In most applications, inference is done online as the user speaks. Providing real-time predictions is essential to user experience [4] and enables new applications like real-time translation [56]. However, inference in speech models can be costly [10] requiring the evaluation of large tensor products in convolutional neural networks.

As users interact with speech services, they provide implicit signal about the quality of the speech predictions which can be used to identify the dialect. Incorporating this feedback quickly improves user experience by allowing us to choose models specialized for a user's dialect.

Benchmark Application: To evaluate the benefit of personalization and online model-selection on a dataset with real user data, we built a speech recognition service with the widely used TIMIT speech corpus [24] and the HTK [63] machine learning framework. This dataset consists of voice recordings for 630 speakers in eight dialects of English. We randomly drew users from the test corpus and simulated their interaction with our speech recognition service using their pre-recorded speech data.

2.2 Challenges

Motivated by the above applications, we outline the key challenges of prediction serving and describe how Clipper addresses these challenges.

Complexity of Deploying Machine Learning

There is a large and growing number of machine learning frameworks [1,7,13,16,31]. Each framework has strengths and weaknesses and many are optimized for specific models or application domains (e.g., computer vision). Thus, there is no dominant framework and often multiple frameworks may be used for a single application (e.g., speech recognition and computer vision in automatic captioning). Furthermore, machine learning is an iterative process and the best framework may change as an application evolves over time (e.g., as a training dataset grows to require distributed model training). Although common model exchange formats have been proposed [47, 48], they have never achieved widespread adoption because of the rapid and fundamental changes in state-of-the-art techniques and additional source of errors from parallel implementations for training and serving. Finally, machine learning frameworks are often developed by and for machine learning experts and are therefore heavily optimized towards model development rather than deployment. As a consequence of these design decisions, application developers are forced to accept reduced accuracy by forgoing the use of a model well-suited to the task or to incur the substantially increased complexity of integrating and supporting multiple machine learning frameworks.

Solution: Clipper introduces a model abstraction layer and common prediction interface that isolates applications from variability in machine learning frameworks (§4) and simplifies the process of deploying a new model or framework to a running application.

Prediction Latency and Throughput

The *prediction latency* is the time it takes to render a prediction given a query. Because prediction serving is often on the critical path, predictions must both be fast and have bounded tail latencies to meet service level objectives [64]. While simple linear models are fast, more sophisticated and often more accurate models such as support vector machines, random forests, and deep neural networks are much more computationally intensive and can have substantial latencies (50-100ms) [13] (see Figure 11 for details). In many cases accuracy can be improved by combining models but at the expense of stragglers and increased tail latencies. Finally, most machine learning frameworks are optimized for offline batch processing and not single-input prediction latency. More-

over, the low and bounded latency demands of interactive applications are often at odds with the design goals of machine learning frameworks.

The computational cost of sophisticated models can substantially impact prediction throughput. For example, a relatively fast neural network which is able to render 100 predictions per second is still orders of magnitude slower than a modern web-server. While batching prediction requests can substantially improve throughput by exploiting optimized BLAS libraries, SIMD instructions, and GPU acceleration it can also adversely affect prediction latency. Finally, under heavy query load it is often preferable to marginally degrade accuracy rather than substantially increase latency or lose availability [3, 23].

Solution: Clipper automatically and adaptively batches prediction requests to maximize the use of batch-oriented system optimizations in machine learning frameworks while ensuring that prediction latency objectives are still met (§4.3). In addition, Clipper employs straggler mitigation techniques to reduce and bound tail latency, enabling model developers to experiment with complex models without affecting serving latency (§5.2.2).

Model Selection

Model development is an iterative process producing many models reflecting different feature representations, modeling assumptions, and machine learning frameworks. Typically developers must decide which of these models to deploy based on offline evaluation using stale datasets or engage in costly online A/B testing. When predictions can influence future queries (e.g., content recommendation), offline evaluation techniques can be heavily biased by previous modeling results. Alternatively, A/B testing techniques [2] have been shown to be statistically inefficient — requiring data to grow *exponentially* in the number of candidate models. The resulting choice of model is typically static and therefore susceptible to changes in model performance due to factors such as feature corruption or concept drift [52]. In some cases the best model may differ depending on the context (e.g., user or region) in which the query originated. Finally, predictions from more than one model can often be combined in ensembles to boost prediction accuracy and provide more robust predictions with confidence bounds.

Solution: Clipper leverages adaptive online model selection and ensembling techniques to incorporate feedback and automatically select and combine predictions from models that can span multiple machine learning frameworks.

2.3 Experimental Setup

Because we include microbenchmarks of many of Clipper's features as we introduce them, we present the experimental setup now. For each of the three object recognition benchmarks, the prediction task is predicting the correct label given the raw pixels of an unlabeled image as input. We used a variety of models on each of the object recognition benchmarks. For the speech recognition benchmark, the prediction task is predicting the phonetic transcription of the raw audio signal. For this benchmark, we used the HTK Speech Recognition Toolkit [63] to learn Hidden Markov Models whose outputs are sequences of phonemes representing the transcription of the sound. Details about each dataset are presented in Table 1.

Dataset	Type	Size	Features	Labels
MNIST [35]	Image	70K	28x28	10
CIFAR [32]	Image	60k	32x32x3	10
ImageNet [49]	Image	1.26M	299x299x3	1000
Speech [24]	Sound	6300	5 sec.	39

Table 1: Datasets. The collection of real-world benchmark datasets used in the experiments.

Unless otherwise noted, all experiments were conducted on a single server. All machines used in the experiments contain 2 Intel Haswell-EP CPUs and 256 GB of RAM running Ubuntu 14.04 on Linux 4.2.0. TensorFlow models were executed on a Nvidia Tesla K20c GPUs with 5 GB of GPU memory and 2496 cores. In the scaling experiment presented in Figure 6, the servers in the cluster were connected with both a 10Gbps and 1Gbps network. For each network, all the servers were located on the same switch. Both network configurations were investigated.

3 System Architecture

Clipper is divided into *model selection* and *model abstraction* layers (see Figure 1). The model abstraction layer is responsible for providing a common prediction interface, ensuring resource isolation, and optimizing the query workload for batch oriented machine learning frameworks. The model selection layer is responsible for dispatching queries to one or more models and combining their predictions based on feedback to improve accuracy, estimate uncertainty, and provide robust predictions.

Before presenting the detailed Clipper system design we first describe the path of a prediction request through the system. Applications issue prediction requests to Clipper through application facing REST or RPC APIs. Prediction requests are first processed by the model selection layer. Based on properties of the prediction request and recent feedback, the model selection layer dispatches the prediction request to one or more of the models through the model abstraction layer.

The model abstraction layer first checks the prediction cache for the query before assigning the query to an adaptive batching queue associated with the desired model. The adaptive batching queue constructs batches of queries that are tuned for the machine learning framework and model. A cross language RPC is used to send the

batch of queries to a model container hosting the model in its native machine learning framework. To simplify deployment, we host each model container in a separate Docker container. After evaluating the model on the batch of queries, the predictions are sent back to the model abstraction layer which populates the prediction cache and returns the results to the model selection layer. The model selection layer then combines one or more of the predictions to render a final prediction and confidence estimate. The prediction and confidence estimate are then returned to the end-user application.

Any feedback the application collects about the quality of the predictions is sent back to the model selection layer through the same application-facing REST/RPC interface. The model selection layer joins this feedback with the corresponding predictions to improve how it selects and combines future predictions.

We now present the model abstraction layer and the model selection layer in greater detail.

4 Model Abstraction Layer

The **Model Abstraction Layer** (Figure 1) provides a common interface across machine learning frameworks. It is composed of a prediction cache, an adaptive query-batching component, and a set of model containers connected to Clipper via a lightweight RPC system. This modular architecture enables caching and batching mechanisms to be shared across frameworks while also scaling to many concurrent models and simplifying the addition of new frameworks.

4.1 Overview

At the top of the model abstraction layer is the prediction cache (§4.2). The prediction caches provides a partial pre-materialization mechanism for frequent queries and accelerates the adaptive model selection techniques described in §5 by enabling efficient joins between recent predictions and feedback.

The batching component (§4.3) sits below the prediction cache and aggregates point queries into mini-batches that are dynamically resized for each model container to maximize throughput. Once a mini-batch is constructed for a given model it is dispatched via the RPC system to the container for evaluation.

Models deployed in Clipper are each encapsulated within their own lightweight container (§4.4), communicating with Clipper through an RPC mechanism that provides a uniform interface to Clipper and simplifies the deployment of new models. The lightweight RPC system minimizes the overhead of the container-based architecture and simplifies cross-language integration.

In the following sections we describe each of these components in greater detail and discuss some of the key algorithmic innovations associated with each.

4.2 Caching

For many applications (e.g., content recommendation), predictions concerning popular items are requested frequently. By maintaining a prediction cache, Clipper can serve these frequent queries without evaluating the model. This substantially reduces latency and system load by eliminating the additional cost of model evaluation.

In addition, caching in Clipper serves an important role in model selection (§5). To select models intelligently Clipper needs to join the original predictions with any feedback it receives. Since feedback is likely to return soon after predictions are rendered [39], even infrequent or unique queries can benefit from caching.

For example, even with a small ensemble of four models (a random forest, logistic regression model, and linear SVM trained in Scikit-Learn and a linear SVM trained in Spark), prediction caching increased feedback processing throughput in Clipper by 1.6x from roughly 6K to 11K observations per second.

The prediction cache acts as a function cache for the generic prediction function:

```
Predict(m: ModelId, x: X) -> y: Y
```

that takes a model id *m* along with the query *x* and computes the corresponding model prediction *y*. The cache exposes a simple non-blocking *request* and *fetch* API. When a prediction is needed, the *request* function is invoked which notifies the cache to compute the prediction if it is not already present and returns a boolean indicating whether the entry is in the cache. The *fetch* function checks the cache and returns the query result if present.

Clipper employs an LRU eviction policy for the prediction cache, using the standard CLOCK [17] cache eviction algorithm. With an adequately sized cache, frequent queries will not be evicted and the cache serves as a partial pre-materialization mechanism for hot items. However, because adaptive model selection occurs *above the cache* in Clipper, changes in predictions due to model selection do not invalidate cache entries.

4.3 Batching

The Clipper batching component transforms the concurrent stream of prediction queries received by Clipper into batches that more closely match the workload assumptions made by machine learning frameworks while simultaneously amortizing RPC and system overheads. Batching improves throughput and utilization of often costly physical resources such as GPUs, but it does so at the expense of increased latency by requiring all queries in the batch to complete before returning a single prediction.

We exploit an explicitly stated latency service level objective (SLO) to *increase latency* in exchange for substantially improved throughput. By allowing users to specify a latency objective, Clipper is able to tune batched query

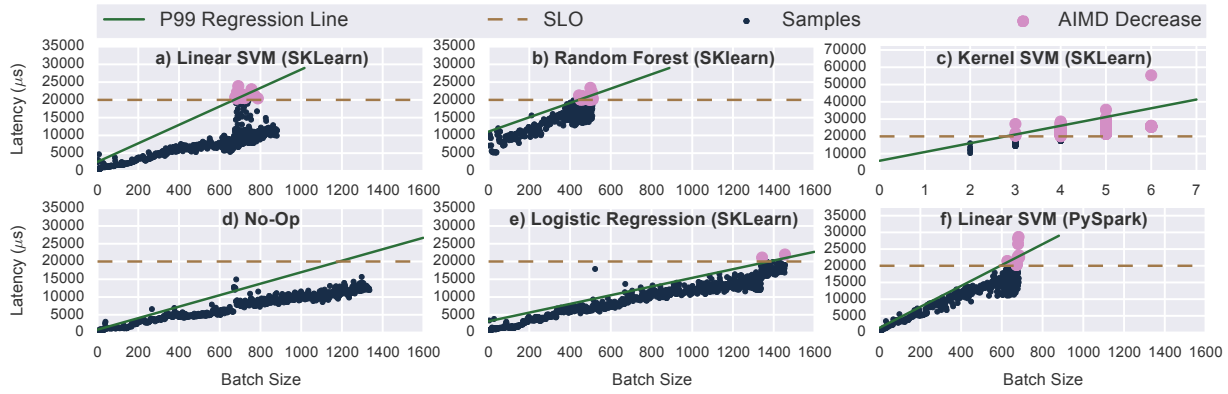


Figure 3: Model Container Latency Profiles. We measured the batching latency profile of several models trained on the MNIST benchmark dataset. The models were trained using Scikit-Learn (SKLearn) or Spark and were chosen to represent several of the most widely used types of models. The No-Op Container measures the system overhead of the model containers and RPC system.

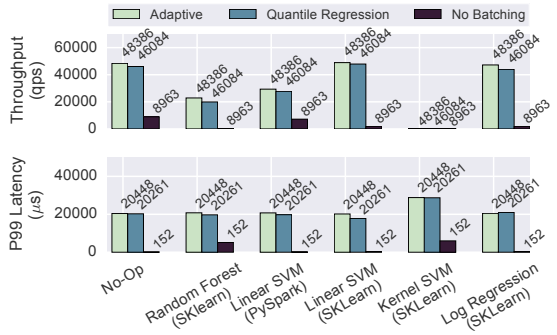


Figure 4: Comparison of Dynamic Batching Strategies.

evaluation to maximize throughput while still meeting the latency requirements of interactive applications. For example, requesting predictions in sufficiently large batches can improve throughput by up to 26x (the Scikit-Learn SVM in Figure 4) while meeting a 20ms latency SLO.

Batching increases throughput via two mechanisms. First, batching amortizes the cost of RPC calls and internal framework overheads such as copying inputs to GPU memory. Second, batching enables machine learning frameworks to exploit existing data-parallel optimizations by performing batch inference on many inputs simultaneously (e.g., by using the GPU or BLAS acceleration).

As the model selection layer dispatches queries for model evaluation, they are placed on queues associated with model containers. Each model container has its own adaptive batching queue tuned to the latency profile of that container and a corresponding thread to process predictions. Predictions are processed in batches by removing as many queries as possible from a queue up to the maximum batch size *for that model container* and sending the queries as a single batch prediction RPC to the container for evaluation. Clipper imposes a *maximum* batch size to ensure that latency objectives are met and avoid excessively delaying the first queries in the batch.

Frameworks that leverage GPU acceleration such as TensorFlow often enforce static batch sizes to maintain a consistent data layout across evaluations of the model. These frameworks typically encode the batch size directly into the model definition in order to fully exploit GPU parallelism. When rendering fewer predictions than the batch size, the input must be padded to reach the defined size, reducing model throughput without any improvement in prediction latency. Careful tuning of the batch size should be done to maximize inference performance, but this tuning must be done offline and is fixed by the time a model is deployed.

However, most machine learning frameworks can efficiently process variable-sized batches at serving time. Yet differences between the framework implementation and choice of model and inference algorithm can lead to orders of magnitude variation in model throughput and latency. As a result, the latency profile – the expected time to evaluate a batch of a given size – varies substantially between model containers. For example, in Figure 3 we see that the maximum batch size that can be executed within a 20ms latency SLO differs by 241x between the linear SVM which does a very simple vector-vector multiply to perform inference and the kernel SVM which must perform a sequence of expensive nearest-neighbor calculations to evaluate the kernel. As a consequence, the linear SVM can achieve throughput of nearly 30,000 qps while the kernel SVM is limited to 200 qps under this SLO. Instead of requiring application developers to manually tune the batch size for each new model, Clipper employs a simple adaptive batching scheme to dynamically find and adapt the maximum batch size.

4.3.1 Dynamic Batch Size

We define the optimal batch size as the batch size that maximizes throughput subject to the constraint that the batch evaluation latency is under the target SLO. To automati-

cally find the optimal maximum batch size for each model container we employ an additive-increase-multiplicative-decrease (AIMD) scheme. Under this scheme, we additively increase the batch size by a fixed amount until the latency to process a batch exceeds the latency objective. At this point, we perform a small multiplicative backoff, reducing the batch size by 10%. Because the optimal batch size does not fluctuate substantially, we use a much smaller backoff constant than other Additive-Increase, Multiplicative-Decrease schemes [15].

Early performance measurements (Figure 3) suggested a stable linear relationship between batch size and latency across several of the modeling frameworks. As a result, we also explored the use of quantile regression to estimate the 99th-percentile (P99) latency as a function of batch size and set the maximum batch size accordingly. We compared the two approaches on a range of commonly used Spark and Scikit-Learn models in Figure 4. Both strategies provide significant performance improvements over the baseline strategy of no batching, achieving up to a 26x throughput increase in the case of the Scikit-Learn linear SVM, demonstrating the performance gains that batching provides. While the two batching strategies perform nearly identically, the AIMD scheme is significantly simpler and easier to tune. Furthermore, the ongoing adaptivity of the AIMD strategy makes it robust to changes in throughput capacity of a model (e.g., during a garbage collection pause in Spark). As a result, Clipper employs the AIMD scheme as the default.

4.3.2 Delayed Batching

Under moderate or bursty loads, the batching queue may contain less queries than the maximum batch size when the next batch is ready to be dispatched. For some models, briefly delaying the dispatch to allow more queries to arrive can significantly improve throughput under bursty loads. Similar to the motivation for Nagle's algorithm [44], the gain in efficiency is a result of the ratio of the fixed cost for sending a batch to the variable cost of increasing the size of a batch.

In Figure 5, we compare the gain in efficiency (measured as increased throughput) from delayed batching for two models. Delayed batching provides no increase in throughput for the Spark SVM because Spark is already relatively efficient at processing small batch sizes and can keep up with the moderate serving workload using batches much smaller than the optimal batch size. In contrast, the Scikit-Learn SVM has a high fixed cost for processing a batch but employs BLAS libraries to do efficient parallel inference on many inputs at once. As a consequence, a 2ms batch delay provides a 3.3x improvement in throughput and allows the Scikit-Learn model container to keep up with the throughput demand while remaining well below the 10-20ms latency objectives needed for interactive applications.

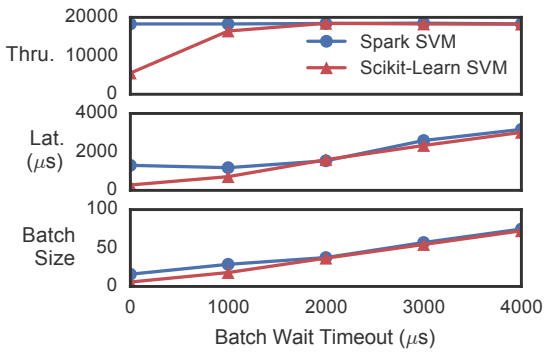


Figure 5: Throughput Increase from Delayed Batching.

```
interface Predictor<X,Y> {
  List<List<Y>> pred_batch(List<X> inputs);
}
```

Listing 1: Common Batch Prediction Interface for Model Containers. The batch prediction function is called via the RPC interface to compute the predictions for a batch of inputs. The return type is a nested list because each input may produce multiple outputs.

4.4 Model Containers

Model containers encapsulate the diversity of machine learning frameworks and model implementations within a uniform "narrow waist" remote prediction API. To add a new type of model to Clipper, model builders only need to implement the standard batch prediction interface in Listing 1. Clipper includes language specific container bindings for C++, Java, and Python. The model container implementations for most of the models in this paper only required a few lines of code.

To achieve process isolation, each model is managed in a separate Docker container. By placing models in separate containers, we ensure that variability in performance and stability of relatively immature state-of-the-art machine learning frameworks does not interfere with the overall availability of Clipper. Any state associated with a model, such as the model parameters, is provided to the container during initialization and the container itself is stateless after initialization. As a result, resource intensive machine learning frameworks can be replicated across multiple machines or given access to specialized hardware (e.g., GPUs) when needed to meet serving demand.

4.4.1 Container Replica Scaling

Clipper supports replicating model containers, both locally and across a cluster, to improve prediction throughput and leverage additional hardware accelerators. Because different replicas can have different performance characteristics, particularly when spread across a cluster, Clipper performs adaptive batching independently for

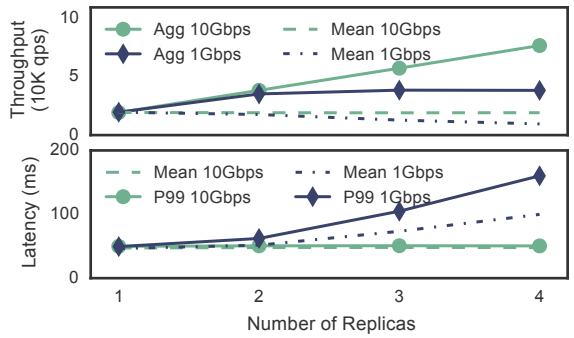


Figure 6: Scaling the Model Abstraction Layer Across a GPU Cluster. The solid lines refer to aggregate throughput of all the model replicas and the dashed lines refer to the mean per-replica throughput.

each replica.

In Figure 6 we demonstrate the linear throughput scaling that Clipper can achieve by replicating model containers across a cluster. With a four-node GPU cluster connected through a 10Gbps Ethernet switch, Clipper gets a 3.95x throughput increase from 19,500 qps when using a single model container running on a local GPU to 77,000 qps when using four replicas each running on a different machine. Because the model containers in this experiment are computationally intensive and run on the GPU, GPU throughput is the bottleneck and Clipper's RPC system can easily saturate the GPUs. However, when the cluster is connected through a 1Gbps switch, the aggregate throughput of the GPUs is higher than 1Gbps and so the network becomes saturated when replicating to a second remote machine. As machine-learning applications begin to consume increasingly bigger inputs, scaling from handcrafted features to large images, audio signals, or even video, the network will continue to be a bottleneck to scaling out prediction serving applications. This suggests the need for research into efficient networking strategies for remote predictions on large inputs.

5 Model Selection Layer

The **Model Selection Layer** uses feedback to dynamically select one or more of the deployed models and combine their outputs to provide more accurate and robust predictions. By allowing many candidate models to be deployed simultaneously and relying on feedback to adaptively determine the best model or combination of models, the model selection layer simplifies the deployment process for new models. By continuously learning from feedback throughout the lifetime of an application, the model selection layer automatically compensates for failing models without human intervention. By combining predictions from multiple models, the model selection layer boosts application accuracy and estimates prediction confidence.

There are a wide range of techniques for model selection and composition that span a tradeoff space of computational overhead and application accuracy. However, most of these techniques can be expressed with a simple *select*, *combine*, and *observe* API. We capture this API in the model selection policy interface (Listing 2) which governs the behavior of the model selection layer and allows users to introduce new model selection techniques themselves.

```
interface SelectionPolicy<S, X, Y> {
  S init();
  List<ModelId> select(S s, X x);
  pair<Y, double> combine(S s, X x,
    Map<ModelId, Y> pred);
  S observe(S s, X x, Y feedback,
    Map<ModelId, Y> pred);
}
```

Listing 2: Model Selection Policy Interface.

The model selection policy (Listing 2) defines four essential functions as well as a few basic types. In addition to the query and prediction types X and Y, the state type S encodes the learned state of the selection algorithm. The *init* function returns an initial instance of the selection policy state. We isolate the selection policy state and require an initialization function to enable Clipper to efficiently instantiate many instances of the selection policy for fine-grained contextualized model selection (§5.3). The *select* and *combine* functions are responsible for choosing which models to query and how to combine the results. In addition, the *combine* function can compute other information about the predictions. For example, in §5.2.1 we leverage the *combine* function to provide a prediction confidence score. Finally, the *observe* function is used to update the state S based on feedback from front-end applications.

In the current implementation of Clipper we provide two generic model selection policies based on robust bandit algorithms developed by Auer et al. [6]. These algorithms span a trade-off between computation overhead and accuracy. The single model selection policy (§5.1) leverages the Exp3 algorithm to optimally *select* the best model based on noisy feedback with minimal computational overhead. The ensemble model selection policy (§5.2) is based on the Exp4 algorithm which adaptively *combines* the predictions to improve prediction accuracy and estimate confidence at the expense of increased computational cost from evaluating all models for each query. By implementing model selection policies that provide different cost-accuracy tradeoffs, as well as an API for users to implement their own policies, Clipper provides a mechanism to easily navigate the tradeoffs between accuracy and computational cost on a per-application basis. Furthermore, users can modify this choice over time as application workloads evolve and resources become more or less constrained.

Framework	Model	Size (Layers)
Caffe	VGG [54]	13 Conv. and 3 FC
Caffe	GoogLeNet [57]	96 Conv. and 5 FC
Caffe	ResNet [29]	151 Conv. and 1 FC
Caffe	CaffeNet [22]	5 Conv. and 3 FC
TensorFlow	Inception [58]	6 Conv, 1 FC, & 3 Incept.

Table 2: Deep Learning Models. The set of deep learning models used to evaluate the ImageNet ensemble selection policy.

5.1 Single Model Selection Policy

We can cast the model-selection process as a multi-armed bandit problem [43]. The multi-armed bandit[1] problem refers the task of optimally choosing between k possible actions (e.g., models) each with a stochastic reward (e.g., feedback). Because only the reward for the *selected* action can be observed, solutions to the multi-armed bandit problem must address the trade-off between *exploring* possible actions and *exploiting* the estimated best action.

There are numerous algorithms for the multi-armed bandits problem with a wide range of trade-offs. In this work we first explore the use of the simple randomized Exp3 [6] algorithm which makes few assumptions about the problem setting and has strong optimality guarantees. The Exp3 algorithm associates a weight $s_i = 1$ for each of the k deployed models and then randomly selects model i with probability $p_i = s_i / \sum_{j=1}^{k} s_j$. For each prediction $\hat{y}$, Clipper observes a loss $L(y,\hat{y}) \in [0,1]$ with respect to the true value y (e.g., the fraction of words that were transcribed correctly during speech recognition). The Exp3 algorithm then updates the weight, $s_i \leftarrow s_i \exp(-\eta L(y,\hat{y})/p_i)$, corresponding to the selected model i. The constant η determines how quickly Clipper responds to recent feedback.

The Exp3 algorithm provides several benefits over manual experimentation and A/B testing, two common ways of performing model-selection in practice. Exp3 is both simple and robust, scaling well to model selection over a large number of models. It is a lightweight algorithm that requires only a single model evaluation for each prediction and thus performs well under heavy loads with negligible computational overhead. And Exp3 has strong theoretical guarantees that ensure it will quickly converge to an optimal solution.

5.2 Ensemble Model Selection Policies

It is a well-known result in machine learning [8,12,30,43] that prediction accuracy can be improved by combining predictions from multiple models. For example, bootstrap aggregation [9] (a.k.a., bagging) is used widely to reduce variance and thereby improve generalization performance. More recently, ensembles were used to win the Netflix challenge [53], and a carefully crafted ensemble of deep neural networks was used to achieve state-of-the-art accuracy on the speech recognition corpus Google uses to power their acoustic models [30]. The ensemble model selection policies adaptively combine the predictions from *all* available models to improve accuracy, rather than select individual models.

[1]The term bandits refers to pull-lever slot machines found in casinos.

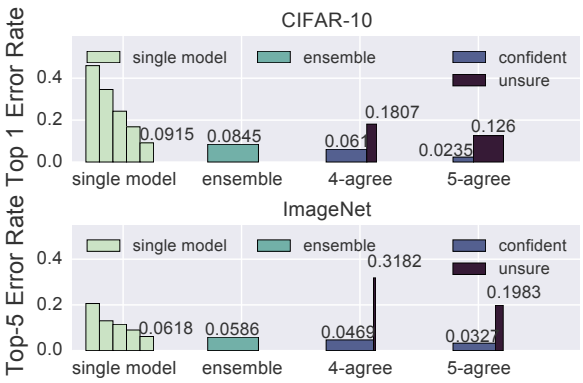


Figure 7: Ensemble Prediction Accuracy. The linear ensembles are composed of five computer vision models (Table 2) applied to the CIFAR and ImageNet benchmarks. The 4-agree and 5-agree groups correspond to ensemble predictions in which the queries have been separated by the ensemble prediction confidence (four or five models agree) and the width of each bar defines the proportion of examples in that category.

In Clipper we use linear ensemble methods which compute a weighted average of the base model predictions. In Figure 7, we show the prediction error rate of linear ensembles on two benchmarks. In both cases linear ensembles are able to marginally reduce the overall error rate. In the ImageNet benchmark, the ensemble formulation achieves a 5.2% relative reduction in the error rate simply by combining off-the-shelf models (Table 2). While this may seem small, on the difficult computer vision tasks for which these models are used, a lot of time and energy is spent trying to achieve even small reductions in error, and marginal improvements are considered significant [49].

There are many methods for estimating the ensemble weights including linear regression, boosting [43], and bandit formulations. We adopt the bandits approach and use the Exp4 algorithm [6] to learn the weights. Unlike Exp3, Exp4 constructs a weighted *combination* of all base model predictions and updates weights based on the individual model prediction error. Exp4 confers many of the same theoretical guarantees as Exp3. But while the accuracy when using Exp3 is bounded by the accuracy of the single best model, Exp4 can further improve prediction accuracy as the number of models increases. The extent to which accuracy increases depends on the relative accuracies of the set of base models, as well as the independence of their predictions. This increased accuracy comes at the cost of increased computational resources consumed by each prediction in order to evaluate all the base models.

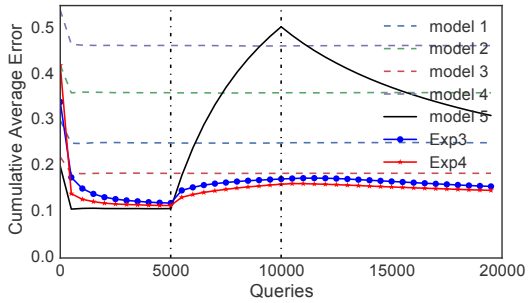


Figure 8: Behavior of Exp3 and Exp4 Under Model Failure. After 5K queries the performance of the lowest-error model is severely degraded, and after 10k queries performance recovers. Exp3 and Exp4 quickly compensate for the failure and achieve lower error than any static model selection.

The accuracy of a deployed model can silently degrade over time. Clipper's online selection policies can automatically detect these failures using feedback and compensate by switching to another model (Exp3) or down-weighting the failing model (Exp4). To evaluate how quickly and effectively the model selection policies react in the presence of changes in model accuracy, we simulated a severe model degradation while receiving real-time feedback. Using the CIFAR dataset we trained five different Caffe models with varying levels of accuracy to perform object recognition. During a simulated run of 20K sequential queries with immediate feedback, we degraded the accuracy of the best-performing model after 5K queries and then allowed the model to recover after 10K queries.

In Figure 8 we plot the cumulative average error rate for each of the five base models as well as the single (Exp3) and ensemble (Exp4) model selection policies. In the first 5K queries both model selection policies quickly converge to an error rate near the best performing model (model 5). When we degrade the predictions from model 5 its cumulative error rate spikes. The model selection policies are able to quickly mitigate the consequences of the increase in errors by learning to divert queries to the other models. When model 5 recovers after 10K queries the model selection policies also begin to improve by gradually sending queries back to model 5.

5.2.1 Robust Predictions

The advantages of online model selection go beyond detecting and mitigating model failures to leveraging new opportunities to improve application accuracy and performance. For many real-time decision-making applications, knowing the confidence of the prediction can significantly improve the end-user experience of the application.

For example, in many settings, applications have a sensible default action they can take when a prediction is unavailable. This is critical for building highly available applications that can survive partial system failures or when building applications where a mistake can be costly. Rather than blindly using all predictions regardless of the confidence in the result, applications can choose to only accept predictions above a confidence threshold by using the robust model selection policy. When the confidence in a prediction for a query falls below the confidence threshold, the application can instead use the sensible default decision for the query and avoid a costly mistake.

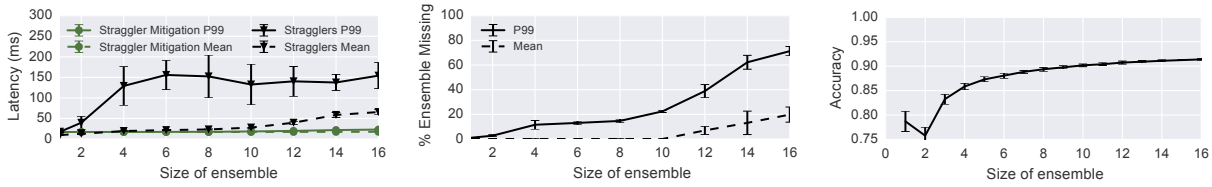
By evaluating predictions from multiple competing models concurrently we can obtain an estimator of the confidence in our predictions. In settings where models have high variance or are trained on random samples from the training data (e.g., bagging), agreement in model predictions is an indicator of prediction confidence. When evaluating the *combine* function in the ensemble selection policy we compute a measure of confidence by calculating the number of models that agree with the final prediction. End user applications can use this confidence score to decide whether to rely on the prediction. If we only consider predictions where multiple models agree, we can substantially reduce the error rate (see Figure 7) while declining to predict a small fraction of queries.

5.2.2 Straggler Mitigation

While the ensemble model selection policy can improve prediction accuracy and help quantify uncertainty, it introduces additional system costs. As we increase the size of the ensemble the computational cost of rendering a prediction increases. Fortunately, we can compensate for the increased prediction cost by scaling-out the model abstraction layer. Unfortunately, as we add model containers we increase the chance of stragglers adversely affecting tail latencies.

To evaluate the cost of stragglers, we deployed ensembles of increasing size and measured the resulting prediction latency (Figure 9a) under moderate query load. Even with small ensembles we observe the effect of stragglers on the P99 tail latency, which rise sharply to well beyond the 20ms latency objective. As the size of the ensemble increases and the system becomes more heavily loaded, stragglers begin to affect the mean latency.

To address stragglers, Clipper introduces a simple best-effort straggler-mitigation strategy motivated by the design choice that rendering a *late* prediction is worse than rendering an *inaccurate* prediction. For each query the model selection layer maintains a latency deadline determined by the latency SLO. At the latency deadline the *combine* function of the model selection policy is invoked with the *subset* of the predictions that are available. The model selection policy must render a final prediction using only the available base model predictions and communicate the potential loss in accuracy in its confidence score. Currently, we substitute missing predictions with their average value and define the confidence as the



(a) Latency **(b)** Missing Predictions **(c)** Accuracy

Figure 9: Increase in stragglers from bigger ensembles. The **(a)** latency, **(b)** percentage of missing predictions, and **(c)** prediction accuracy when using the ensemble model selection policy on SK-Learn Random Forest models applied to MNIST. As the size of an ensemble grows, the prediction accuracy increases but the latency cost of blocking until all predictions are available grows substantially. Instead, Clipper enforces bounded latency predictions and transforms the latency cost of waiting for stragglers into a reduction in accuracy from using a smaller ensemble.

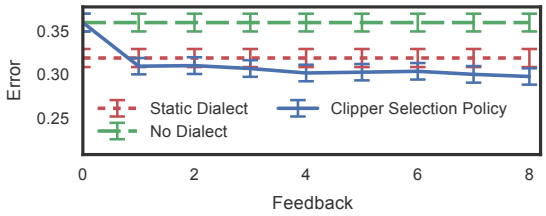


Figure 10: Personalized Model Selection. Accuracy of the ensemble selection policy on the speech recognition benchmark.

fraction of models that agree on the prediction.

The best-effort straggler-mitigation strategy prevents model container tail latencies from propagating to front-end applications by maintaining the latency objective as additional models are deployed. However, the straggler mitigation strategy reduces the size of the ensemble. In Figure 9b we plot the reduction in ensemble size and find that while tail latencies increase significantly with even small ensembles, most of the predictions arrive by the latency deadline. In Figure 9c we plot the effect of ensemble size on accuracy and observe that this ensemble can tolerate the loss of small numbers of component models with only a slight reduction in accuracy.

5.3 Contextualization

In many prediction tasks the accuracy of a particular model may depend heavily on context. For example, in speech recognition a model trained for one dialect may perform well for some users and poorly for others. However, selecting the right model or composition of models can be difficult and is best accomplished online in the model selection layer through feedback. To support context specific model selection, the model selection layer can be configured to instantiate a unique model selection state for each user, context, or session. The context specific session state is managed in an external database system. In our current implementation we use Redis.

To demonstrate the potential gains from personalized model selection we hosted a collection of TIMIT [24] voice recognition models each trained for a different dialect. We then evaluated (Figure 10) the prediction error rates using a single model trained across all dialects, the users' reported dialect model, and the Clipper ensemble selection policy. We first observe that the dialect-specific models out-perform the dialect-oblivious model, demonstrating the value of context to improve prediction accuracy. We also observe that the ensemble selection policy is able to quickly identify a combination of models that out-performs even the users' designated dialect model by using feedback from the serving workload.

6 System Comparison

In addition to the microbenchmarks presented in §4 and §5, we compared Clipper's performance to TensorFlow Serving and evaluate latency and throughput on three object recognition benchmarks.

TensorFlow Serving [59] is a recently released prediction serving system created by Google to accompany their TensorFlow machine learning training framework. Similar to Clipper, TensorFlow Serving is designed for serving machine learning models in production environments and provides a high-performance prediction API to simplify deploying new algorithms and experimenting with new models without modifying frontend applications. TensorFlow Serving supports general TensorFlow models with GPU acceleration through direct integration with the TensorFlow machine learning framework and tightly couples the model and serving components in the same process.

TensorFlow Serving also employs batching to accelerate prediction serving. Batch sizes in TensorFlow Serving are static and rely on a purely timeout based mechanism to avoid starvation. TensorFlow Serving does not explicitly incorporate prediction latency objectives which must be achieved by manually tuning the batch size. Furthermore, TensorFlow Serving was designed to serve one model at a time and therefore does not directly support feedback, dynamic model selection, or composition.

To better understand the performance overheads intro-

duced by Clipper's layered architecture and decoupled model containers, we compared the serving performance of Clipper and TensorFlow Serving on three TensorFlow object recognition deep networks of varying computational cost: a 4-layer convolutional neural network trained on the MNIST dataset [42], the 8-layer AlexNet [33] architecture trained on CIFAR-10 [32], and Google's 22-layer Inception-v3 network [58] trained on ImageNet. We implemented two Clipper model containers for each TensorFlow model, one that calls TensorFlow from the more standard and widely used Python API and one that calls TensorFlow from the more efficient C++ API. All models were run on a GPU using hand-tuned batch sizes (MNIST: 512, CIFAR: 128, ImageNet: 16) to maximize the throughput of TensorFlow Serving. The serving workload measured the maximum sustained throughput and corresponding prediction latency for each system.

Despite Clipper's modular design, we are able to achieve comparable throughput to TensorFlow Serving across all three models (Figure 11). The Python model containers suffer a 15-18% performance hit compared to the throughput of TensorFlow Serving, but the C++ model containers achieve nearly identical performance. This suggests that the high-level Python API for TensorFlow imposes a significant performance cost in the context of low-latency prediction-serving but that Clipper does not impose any additional performance degradation.

For these serving workloads, the throughput bottleneck is inference on the GPU. Both systems utilize additional queuing in order to saturate the GPU and therefore maximize throughput. For the Clipper model containers, we decomposed the prediction latency into component functions to demonstrate the overhead of the modular system design. The *predict* bar is the time spent performing inference within TensorFlow framework code. The *queue* bar is time spent queued within the model container waiting for the GPU to become available. The top bar includes the remaining system overhead, including query serialization and deserialization as well as copying into and out of the network stack. As Figure 11 illustrates, the RPC overheads are minimal on these workloads and the next prediction batch is queued as soon as the current batch is dispatched to the GPU for inference. TensorFlow Serving utilizes a similar queueing method to saturate the GPU, but because of the tight integration between TensorFlow Serving and the TensorFlow inference code, they are able to push the queueing into the TensorFlow framework code itself running in the same process.

By achieving comparable performance across this range of models, we have demonstrated that through careful design and implementation of the system, the modular architecture and substantially broader set of features in Clipper do not come at a cost of reduced performance on core prediction-serving tasks.

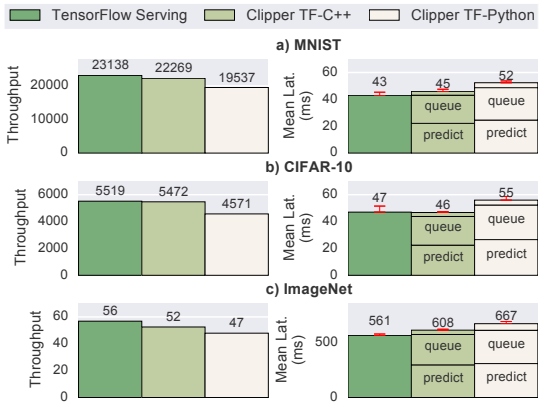


Figure 11: TensorFlow Serving Comparison. Comparison of peak throughput and latency (p99 latencies shown in error bars) on three TensorFlow models of varying inference cost. TF-C++ uses TensorFlow's C++ API and TF-Python the Python API.

7 Limitations

While Clipper attempts to address many challenges in the context of prediction serving there are a few key limitations when compared to other designs like TensorFlow Serving. Most of these limitations follow directly from the design of the Clipper architecture which assumes models are below Clipper in the software stack, and thus are treated as black-box components.

Clipper does not optimize the execution of the models within their respective machine learning frameworks. Slow models will remain slow when served from Clipper. In contrast, TensorFlow Serving is tightly integrated with model evaluation, and hence is able to leverage GPU acceleration and compilation techniques to speedup inference on models created with TensorFlow.

Similarly, Clipper does not manage the training or retraining of the base models within their respective frameworks. As a consequence, if all models are out-of-date or inaccurate Clipper will be unable to improve accuracy beyond what can be accomplished through ensembles.

8 Related Work

The closest projects to Clipper are LASER [3], Velox [19], and TensorFlow Serving [59]. The LASER system was developed at LinkedIn to support linear models for ad-targeting applications. Velox is a UC Berkeley research project to study personalized prediction serving with Apache Spark. TensorFlow Serving is the open-source prediction serving system developed by Google for TensorFlow models. In our experiments we only compare against TensorFlow Serving, because LASER is not publicly available, and the current prototype of Velox has very limited functionality.

All three systems propose mechanisms to address latency and throughput. Both LASER and Velox utilize

caching at various levels in their systems. In addition, LASER also uses a straggler mitigation strategy to address slow feature evaluation. Neither LASER or Velox discuss batching. Conversely, TensorFlow Serving does not employ caching and instead leverages batching and hardware acceleration to improve throughput.

LASER and Velox both exploit a form of model decomposition to incorporate feedback and context similar to the linear ensembles in Clipper. However, LASER does not incorporate feedback in real-time, Velox does not support bandits and neither system supports cross framework learning. Moreover, the techniques used for online learning and contextualization in both of these systems are captured in the more general Clipper selection policy. In contrast, TensorFlow Serving has no mechanism to achieve personalization or adapt to real-time feedback.

Finally, LASER, Velox, and TensorFlow Serving are all vertically integrated; they focused on serving predictions from a single model or framework. In contrast, Clipper supports a wide range of machine learning models and frameworks and simultaneously addresses latency, throughput, and accuracy in a single serving system.

Application Specific Prediction Serving: There has been considerable prior work in application and model specific prediction-serving. Much of this work has focused on content recommendation, including video-recommendation [20], ad-targeting [27, 39], and product-recommendations [37]. Outside of content recommendation, there has been recent success in speech recognition [36, 55] and internet-scale resource allocation [23]. While many of these applications require real-time predictions, the solutions described are highly application-specific and tightly coupled to the model and workload characteristics. As a consequence, much of this work solves the same systems challenges in different application areas. In contrast, Clipper is a general-purpose system capable of serving many of these applications.

Parameter Server: There has been considerable work in the learning systems community on parameter-servers [5, 21, 38, 62]. While parameter-servers do focus on reduced latency and caching, they do so in the context of *model training*. In particular they are a specialized type of key-value store used to coordinate updates to model parameters in a distributed training system. They are not typically used to serve predictions.

General Serving Systems: The high-performance serving architecture of Clipper draws from prior work on highly-concurrent serving systems [45, 46, 50, 61]. The division of functionality into vertical stages introduced by [61] is similar to the division of Clipper's architecture into independent layers. Notably, while the dominant cost in data-serving systems tends to be IO, in prediction serving it is computation. This changes both physical resource allocation and batching and latency-hiding strategies.

9 Conclusion

In this work we identified three key challenges of prediction serving: latency, throughput, and accuracy, and proposed a new layered architecture that addresses these challenges by interposing between end-user applications and existing machine learning frameworks.

As an instantiation of this architecture, we introduced the Clipper prediction serving system. Clipper isolates end-user applications from the variability and diversity in machine learning frameworks by providing a common prediction interface. As a consequence, new machine learning frameworks and models can be introduced without modifying end-user applications.

We addressed the challenges of prediction serving latency and throughput within the Clipper Model Abstraction layer. The model abstraction layer lifts caching and adaptive batching strategies above the machine learning frameworks to achieve up to a 26x improvement in throughput while maintaining strict bounds on tail latency and providing mechanisms to scale serving across a cluster. We addressed the challenges of accuracy in the Clipper Model Selection Layer. The model selection layer enables many models to be deployed concurrently and then dynamically selects and combines predictions from each model to render more robust, accurate, and contextualized predictions while mitigating the cost of stragglers.

We evaluated Clipper using four standard machine-learning benchmark datasets spanning computer vision and speech recognition applications. We demonstrated Clipper's capacity to bound latency, scale heavy workloads across nodes, and provide accurate, robust, and contextual predictions. We compared Clipper to Google's TensorFlow Serving system and achieved parity on throughput and latency performance, demonstrating that the modular container-based architecture and substantial additional functionality in Clipper can be achieved with minimal performance penalty.

Acknowledgments

We would like to thank Peter Bailis, Alexey Tumanov, Noah Fiedel, Chris Olston, our shepherd Mike Dahlin, and the anonymous reviewers for their feedback. This research is supported in part by DHS Award HSHQDC-16-3-00083, DOE Award SN10040 DE-SC0012463, NSF CISE Expeditions Award CCF-1139158, and gifts from Ant Financial, Amazon Web Services, CapitalOne, Ericsson, GE, Google, Huawei, Intel, IBM, Microsoft and VMware.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015. *Software available from tensorflow. org*.

[2] A. Agarwal, S. Bird, M. Cozowicz, L. Hoang, J. Langford, S. Lee, J. Li, D. Melamed, G. Oshri, O. Ribas, et al. A multiworld testing decision service. *arXiv preprint arXiv:1606.03966*, 2016.

[3] D. Agarwal, B. Long, J. Traupman, D. Xin, and L. Zhang. Laser: A scalable response prediction platform for online advertising. In *WSDM*, pages 173–182, 2014.

[4] S. Agarwal and J. R. Lorch. Matchmaking for online games and other latency-sensitive p2p systems. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 315–326. ACM, 2009.

[5] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *WSDM*, pages 123–132, 2012.

[6] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM J. Comput.*, 32(1):48–77, Jan. 2003.

[7] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, page 3. Austin, TX, 2010.

[8] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag, Secaucus, NJ, USA, 2006.

[9] L. Breiman. Bagging predictors. *Mach. Learn.*, 24(2):123–140, Aug. 1996.

[10] C. Chelba, D. Bikel, M. Shugrina, P. Nguyen, and S. Kumar. Large scale language modeling in automatic speech recognition. *arXiv preprint arXiv:1210.8440*, 2012.

[11] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting Distributed Synchronous SGD. *arXiv.org*, Apr. 2016.

[12] T. Chen and C. Guestrin. XGBoost: A Scalable Tree Boosting System. *arXiv.org*, Mar. 2016.

[13] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[14] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, pages 571–582, 2014.

[15] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Comput. Netw. ISDN Syst.*, 17(1):1–14, June 1989.

[16] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[17] F. J. Corbato. A paging experiment with the multics system. 1968.

[18] Microsoft Cortana. `https://www.microsoft.com/en-us/mobile/experiences/cortana/`.

[19] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. In *CIDR 2015*, 2015.

[20] J. Davidson, B. Liebald, J. Liu, P. Nandy, T. Van Vleet, U. Gargi, S. Gupta, Y. He, M. Lambert, B. Livingston, and D. Sampath. The YouTube video recommendation system. *RecSys*, pages 293–296, 2010.

[21] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. aurelio Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, pages 1223–1231. 2012.

[22] J. Donahue. Caffenet. `https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet`.

[23] A. Ganjam, F. Siddiqui, J. Zhan, X. Liu, I. Stoica, J. Jiang, V. Sekar, and H. Zhang. C3: Internet-Scale Control Plane for Video Quality Optimization. *NSDI '15*, pages 131–144, 2015.

[24] J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, D. S. Pallett, and N. L. Dahlgren. Darpa timit acoustic phonetic continuous speech corpus cdrom, 1993.

[25] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. OSDI, pages 17–30, 2012.

[26] Google Now. `https://www.google.com/landing/now/`.

[27] T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich. Web-Scale Bayesian Click-Through rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine. *ICML*, pages 13–20, 2010.

[28] h20. `http://www.h2o.ai`.

[29] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[30] G. Hinton, O. Vinyals, and J. Dean. Distilling the Knowledge in a Neural Network. *arXiv.org*, Mar. 2015.

[31] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pages 675–678. ACM, 2014.

[32] A. Krizhevsky and G. Hinton. Cifar-10 dataset. `https://www.cs.toronto.edu/~kriz/cifar.html`.

[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1097–1105, 2012.

[34] J. Langford, L. Li, and A. Strehl. Vowpal wabbit online learning project, 2007.

[35] Y. LeCun, C. Cortes, and C. J. Burges. MNIST handwritten digit database. 1998.

[36] X. Lei, A. W. Senior, A. Gruenstein, and J. Sorensen. Accurate and compact large vocabulary speech recognition on mobile devices. *INTERSPEECH*, pages 662–665, 2013.

[37] R. Lerallut, D. Gasselin, and N. Le Roux. Large-Scale Real-Time Product Recommendation at Criteo. In *RecSys*, pages 232–232, 2015.

[38] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, 2014.

[39] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, S. Chikkerur, D. Liu, M. Wattenberg, A. M. Hrafnkelsson, T. Boulos, and J. Kubica. Ad click prediction: a view from the trenches. In *KDD*, page 1222, 2013.

[40] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

[41] V. Mnih, N. Heess, A. Graves, et al. Recurrent models of visual attention. In *NIPS*, pages 2204–2212, 2014.

[42] Deep MNIST for Experts. `https://www.tensorflow.org/versions/r0.10/tutorials/mnist/pros/index.html`.

[43] K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[44] J. Nagle. Congestion control in ip/tcp internetworks. *SIGCOMM Comput. Commun. Rev.*, 14(4):11–17, Oct. 1984.

[45] nginx [engine x]. `http://nginx.org/en/`.

[46] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. *USENIX Annual Technical Conference, General Track*, pages 199–212, 1999.

[47] Portable Format for Analytics (PFA). `http://dmg.org/pfa/index.html`.

[48] PMML 4.2. `http://dmg.org/pmml/v4-2-1/GeneralStructure.html`.

[49] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[50] D. C. Schmidt. Pattern languages of program design. chapter Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching, pages 529–545. 1995.

[51] Scikit-Learn machine learning in python. `http://scikit-learn.org`.

[52] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison. Hidden Technical Debt in Machine Learning Systems. *NIPS*, 2015.

[53] J. Sill, G. Takács, L. Mackey, and D. Lin. Feature-weighted linear stacking, 2009.

[54] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[55] Apple Siri. `http://www.apple.com/ios/siri/`.

[56] Skype real time translator. `https://www.skype.com/en/features/skype-translator/`.

[57] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *CVPR*, pages 1–9, 2015.

[58] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. *arXiv preprint arXiv:1512.00567*, 2015.

[59] TensorFlow Serving. `https://tensorflow.github.io/serving`.

[60] Turi. `https://turi.com`.

[61] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *SOSP*, pages 230–243, 2001.

[62] E. P. Xing, Q. Ho, W. Dai, J.-K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. In *KDD*, pages 1335–1344. ACM, 2015.

[63] S. J. Young, G. Evermann, M. J. F. Gales, T. Hain, D. Kershaw, G. Moore, J. Odell, D. Ollason, D. Povey, V. Valtchev, and P. C. Woodland. *The HTK Book, version 3.4*. Cambridge University Engineering Department, Cambridge, UK, 2006.

[64] J.-M. Yun, Y. He, S. Elnikety, and S. Ren. Optimal aggregation policy for reducing tail latency of web search. In *SIGIR*, pages 63–72, 2015.

[65] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds

Kevin Hsieh[†] Aaron Harlap[†] Nandita Vijaykumar[†] Dimitris Konomis[†]
Gregory R. Ganger[†] Phillip B. Gibbons[†] Onur Mutlu[§†]
[†]Carnegie Mellon University [§]ETH Zürich

Abstract

Machine learning (ML) is widely used to derive useful information from large-scale data (such as user activities, pictures, and videos) generated at increasingly rapid rates, *all over the world*. Unfortunately, it is infeasible to move all this globally-generated data to a centralized data center before running an ML algorithm over it—moving large amounts of raw data over wide-area networks (WANs) can be extremely slow, and is also subject to the constraints of privacy and data sovereignty laws. This motivates the need for a geo-distributed ML system spanning *multiple data centers*. Unfortunately, communicating over WANs can significantly degrade ML system performance (by as much as 53.7× in our study) because the communication overwhelms the limited WAN bandwidth.

Our goal in this work is to develop a *geo-distributed* ML system that (1) employs an intelligent communication mechanism over WANs to efficiently utilize the scarce WAN bandwidth, while retaining the accuracy and correctness guarantees of an ML algorithm; and (2) is generic and flexible enough to run a wide range of ML algorithms, without requiring any changes to the algorithms.

To this end, we introduce a new, general geo-distributed ML system, *Gaia*, that decouples the communication *within* a data center from the communication *between* data centers, enabling different communication and consistency models for each. We present a new ML synchronization model, *Approximate Synchronous Parallel (ASP)*, whose key idea is to dynamically eliminate *insignificant* communication between data centers while still guaranteeing the correctness of ML algorithms. Our experiments on our prototypes of Gaia running across 11 Amazon EC2 global regions and on a cluster that emulates EC2 WAN bandwidth show that Gaia provides 1.8–53.5× speedup over two state-of-the-art distributed ML systems, and is within 0.94–1.40× of the speed of running the same ML algorithm on machines on a local area network (LAN).

1. Introduction

Machine learning (ML) is very widely used across a variety of domains to extract useful information from large-scale data. It has many classes of applications such as image or video classification (e.g., [24,39,65]), speech recognition (e.g., [4]), and topic modeling (e.g., [10]). These applications analyze massive amounts of data from user activities, pictures, videos, etc., which are generated at very rapid rates, *all over the world*. Many large organizations, such as Google [28], Microsoft [51], and Amazon [7], operate tens of data centers globally to minimize their service latency to end-users, and store massive quantities of data all over the globe [31, 33, 36, 41, 57, 58, 71–73, 76].

A commonly-used approach to run an ML application over such rapidly generated data is to *centralize* all data into *one data center* over wide-area networks (WANs) before running the ML application [9,12,44,68]. However, this approach can be prohibitively difficult because: (1) WAN bandwidth is a scarce resource, and hence moving all data can be extremely slow [12,57]. Furthermore, the fast growing rate of image and video generation will eventually saturate the total WAN bandwidth, whose growth has been decelerating for many years [67, 73]. (2) Privacy and data sovereignty laws in some countries prohibit transmission of *raw data* across national or continental borders [12, 72, 73].

This motivates the need to *distribute* an ML system across *multiple data centers*, globally. In such a system, large amounts of raw data are stored locally in different data centers, and the ML algorithms running over the distributed data communicate between data centers using WANs. Unfortunately, existing large-scale distributed ML systems [5, 13, 45, 47, 50, 77] are suitable only for data residing *within* a single data center. Our experiments using three state-of-the-art distributed ML systems (Bösen [74], IterStore [17], and GeePS [18]) show that operating these systems across as few as two data centers (over WANs) can cause a slowdown of 1.8–53.7× (see Section 2.3 and Section 6) relative to their performance within a data center (over LANs). Existing systems that do address challenges in geo-distributed data analytics [12, 33, 36, 41, 57, 58, 71–73] do *not* consider the broad class of important, sophisticated ML algorithms commonly run on ML systems — they focus instead on other types of computation, e.g., map-reduce or SQL.

Our goal in this work is to develop a geo-distributed ML system that (1) minimizes communication over WANs, so that the system is not bottlenecked by the scarce WAN bandwidth; and (2) is general enough to be applicable to a wide variety of ML algorithms, without requiring any changes to the algorithms themselves.

To achieve these goals, such a system needs to address two key challenges. First, to efficiently utilize the limited (and heterogeneous) WAN bandwidth, we need to find an effective communication model that minimizes communication over WANs but still retains the correctness guarantee for an ML algorithm. This is difficult because ML algorithms typically require extensive communication to exchange updates that keep the global ML model sufficiently consistent across data centers. These updates are

required to be timely, irrespective of the available network bandwidth, to ensure algorithm correctness. Second, we need to design a *general* system that effectively handles WAN communication for ML algorithms without requiring any algorithm changes. This is challenging because the communication patterns vary significantly across different ML algorithms [37, 54, 60, 64, 66, 69]. Altering the communication across systems can lead to different tradeoffs and consequences for different algorithms [83].

In this work, we introduce *Gaia*, a new general, geo-distributed ML system that is designed to efficiently operate over a collection of data centers. Gaia builds on the widely used *parameter server* architecture (e.g., [5, 6, 13, 16, 17, 20, 34, 45, 74, 77]) that provides ML worker machines with a distributed global shared memory abstraction for the ML model parameters they collectively train until *convergence* to fit the input data. The key idea of Gaia is to maintain an approximately-correct copy of the global ML model *within* each data center, and dynamically eliminate any unnecessary communication *between* data centers. Gaia enables this by *decoupling* the synchronization (i.e., communication/consistency) model within a data center from the synchronization model between different data centers. This differentiation allows Gaia to run a conventional synchronization model [19, 34, 74] that maximizes utilization of the more-freely-available LAN bandwidth *within* a data center. At the same time, across different data centers, Gaia employs a new synchronization model, called *Approximate Synchronous Parallel (ASP)*, which makes more efficient use of the scarce and heterogeneous WAN bandwidth. By ensuring that each ML model copy in different data centers is *approximately correct* based on a precise notion defined by ASP, we guarantee ML algorithm convergence.

ASP is based on a key finding that the vast majority of updates to the global ML model parameters from each ML worker machine are *insignificant*. For example, our study of three classes of ML algorithms shows that more than 95% of the updates produce less than a 1% change to the parameter value. With ASP, these insignificant updates to the same parameter within a data center are *aggregated* (and thus not communicated to other data centers) until the aggregated updates are significant enough. ASP allows the ML programmer to specify the *function* and the *threshold* to determine the significance of updates for each ML algorithm, while providing default configurations for unmodified ML programs. For example, the programmer can specify that all updates that produce more than a 1% change are significant. ASP ensures all significant updates are synchronized across all model copies in a timely manner. It dynamically adapts communication to the available WAN bandwidth between pairs of data centers and uses special *selective barrier* and *mirror clock* control messages to ensure algorithm convergence even during a period of sudden fall (negative spike) in available WAN bandwidth.

In contrast to a state-of-the-art communication-efficient synchronization model, Stale Synchronous Parallel (SSP) [34], which bounds how *stale (i.e., old)* a parameter can be, ASP bounds how *inaccurate* a parameter can be, in comparison to the most up-to-date value. Hence, it provides high flexibility in performing (or not performing) updates, as the server can delay synchronization *indefinitely* as long as the aggregated update is insignificant.

We build two prototypes of Gaia on top of two state-of-the-art parameter server systems, one specialized for CPUs [17] and another specialized for GPUs [18]. We deploy Gaia across 11 regions on Amazon EC2, and on a local cluster that emulates the WAN bandwidth across different Amazon EC2 regions. Our evaluation with three popular classes of ML algorithms shows that, compared to two state-of-the-art parameter server systems [17, 18] deployed on WANs, Gaia: (1) significantly improves performance, by 1.8–53.5×, (2) has performance within 0.94–1.40× of running the same ML algorithm on a LAN in a single data center, and (3) significantly reduces the monetary cost of running the same ML algorithm on WANs, by 2.6–59.0×.

We make three major contributions:

- To our knowledge, this is the first work to propose a general geo-distributed ML system that (1) differentiates the communication over a LAN from the communication over WANs to make efficient use of the scarce and heterogeneous WAN bandwidth, and (2) is general and flexible enough to deploy a wide range of ML algorithms while requiring *no* change to the ML algorithms themselves.
- We propose a new, efficient ML synchronization model, Approximate Synchronous Parallel (ASP), for communication between parameter servers across data centers over WANs. ASP guarantees that each data center's view of the ML model parameters is approximately the same as the "fully-consistent" view and ensures that all significant updates are synchronized in time. We prove that ASP provides a theoretical guarantee on algorithm convergence for a widely used ML algorithm, stochastic gradient descent.
- We build two prototypes of our proposed system on CPU-based and GPU-based ML systems, and we demonstrate their effectiveness over 11 globally distributed regions with three popular ML algorithms. We show that our system provides significant performance improvements over two state-of-the-art distributed ML systems [17,18], and significantly reduces the communication overhead over WANs.

2. Background and Motivation

We first introduce the architectures of widely-used distributed ML systems. We then discuss WAN bandwidth constraints and study the performance implications of running two state-of-the-art ML systems over WANs.

2.1. Distributed Machine Learning Systems

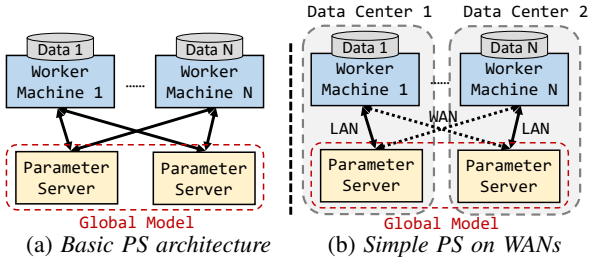
While ML algorithms have different types across different domains, almost all have the same goal—searching for

the best *model* (usually a set of *parameters*) to describe or explain the input *data* [77]. For example, the goal of an image classification neural network is to find the parameters (of the neural network) that can most accurately classify the input images. Most ML algorithms iteratively refine the ML model until it *converges* to fit the data. The correctness of an ML algorithm is thus determined by whether or not the algorithm can *accurately converge* to the best model for its input data.

As the input data to an ML algorithm is usually enormous, processing all input data on a single machine can take an unacceptably long time. Hence, the most common strategy to run a large-scale ML algorithm is to distribute the input data among *multiple* worker machines, and have each machine work on a *shard* of the input data in parallel with other machines. The worker machines communicate with each other periodically to *synchronize* the updates from other machines. This strategy, called *data parallelism* [14], is widely used in many popular ML systems (e.g., [1, 2, 5, 13, 45, 47, 50, 77]).

There are many large-scale distributed ML systems, such as ones using the MapReduce [14] abstraction (e.g., MLlib [2] and Mahout [1]), ones using the graph abstraction (e.g., GraphLab [47] and PowerGraph [26]), and ones using the parameter server abstraction (e.g., Petuum [77] and TensorFlow [5]). Among them, the parameter server architecture provides a performance advantage[1] over other systems for many ML applications and has been widely adopted in many ML systems.

Figure 1a illustrates the high-level overview of the parameter server (PS) architecture. In such an architecture, each parameter server keeps a shard of the global model parameters as a key-value store, and each worker machine communicates with the parameter servers to READ and UPDATE the corresponding parameters. The major benefit of



(a) *Basic PS architecture* (b) *Simple PS on WANs*

Figure 1: Overview of the parameter server architecture

Synchronization among workers in a distributed ML system is a critical operation. Each worker needs to see other workers' updates to the global model to compute more accurate updates using fresh information. However, synchronization is a high-cost operation that can significantly slow down the workers and reduce the benefits of parallelism. The trade-off between *fresher updates* and *communication overhead* leads to three major synchronization models: (1) **Bulk Synchronous Parallel (BSP)** [70], which synchronizes all updates after each worker goes through its shard of data; all workers need to see the most up-to-date model before proceeding to the next iteration, (2) **Stale Synchronous Parallel (SSP)** [34], which allows the fastest worker to be ahead of the slowest worker by up to a bounded number of iterations, so the fast workers may proceed with a *bounded stale* (i.e., old) model, and (3) **Total Asynchronous Parallel (TAP)** [59], which removes the synchronization between workers completely; all workers keep running based on the results of best-effort communication (i.e., each sends/receives as many updates as possible). Both BSP and SSP guarantee algorithm convergence [19, 34], while there is no such guarantee for TAP. Most state-of-the-art parameter servers implement both BSP and SSP (e.g., [5, 16–18, 34, 45, 77]).

As discussed in Section 1, many ML applications need to analyze geo-distributed data. For instance, an image classification system would use pictures located at different data centers as its input data to keep improving its classification using the pictures generated continuously all over the world. Figure 1b depicts the straightforward approach to achieve this goal. In this approach, the worker machines in each data center (i.e., within a LAN) handle the input data stored in the corresponding data center. The parameter servers are evenly distributed across multiple data centers. Whenever the communication between a worker machine and a parameter server crosses data centers, it does so on WANs.

2.2. WAN Network Bandwidth and Cost

WAN bandwidth is a very scarce resource [42, 58, 73] relative to LAN bandwidth. Moreover, the high cost of adding network bandwidth has resulted in a deceleration of WAN bandwidth growth. The Internet capacity growth has fallen steadily for many years, and the annual growth rates have lately settled into the low-30 percent range [67].

To quantify the scarcity of WAN bandwidth between data centers, we measure the network bandwidth between all pairs of Amazon EC2 sites in 11 different regions (Virginia, California, Oregon, Ireland, Frankfurt, Tokyo, Seoul, Singapore, Sydney, Mumbai, and São Paulo). We use `iperf3` [23] to measure the network bandwidth of each pair of different regions for five rounds, and then calculate the average bandwidth. Figure 2 shows the average network bandwidth between each pair of different regions. We make two observations.

First, the WAN bandwidth between data centers is 15× smaller than the LAN bandwidth within a data center on average, and up to 60× smaller in the worst case (for Singapore ↔ São Paulo). Second, the WAN bandwidth *varies significantly* between different regions. The WAN bandwidth between geographically-close regions (e.g., Oregon ↔ California or Tokyo ↔ Seoul) is up to 12× of the bandwidth between distant regions (e.g., Singapore

[1]For example, a state-of-the-art parameter server, IterStore [17], is shown to outperform PowerGraph [26] by 10× for Matrix Factorization. In turn, PowerGraph is shown to match the performance of GraphX [27], a Spark [79] based system.

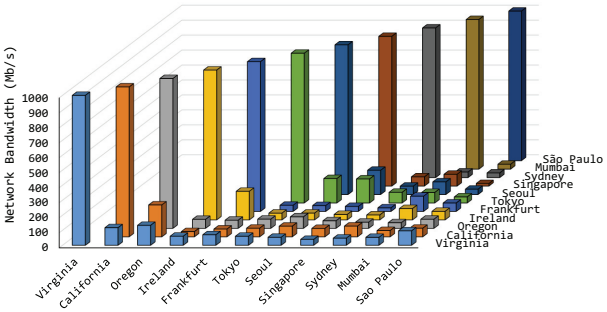


Figure 2: Measured network bandwidth between Amazon EC2 sites in 11 different regions

↔ São Paulo). As Section 2.3 shows, the scarcity and variation of the WAN bandwidth can significantly degrade the performance of state-of-the-art ML systems.

Another important challenge imposed by WANs is the *monetary cost* of communication. In data centers, the cost of WANs far exceeds the cost of a LAN and makes up a significant fraction of the overall cost [29]. Cloud service providers, such as Amazon EC2, charge an extra fee for WAN communication while providing LAN communication free of charge. The cost of WAN communication can be much higher than the cost of the machines themselves. For example, the cost of two machines in Amazon EC2 communicating at the rate of the average WAN bandwidth between data centers is up to 38× of the cost of renting these two machines [8]. These costs make running ML algorithms on WANs much more expensive than running them on a LAN.

2.3. ML System Performance on WANs

We study the performance implications of deploying distributed ML systems on WANs using two state-of-the-art parameter server systems, IterStore [17] and Bösen [74]. Our experiments are conducted on our local 22-node cluster that emulates the WAN bandwidth between Amazon EC2 data centers, the accuracy of which is validated against a real Amazon EC2 deployment (see Section 5.1 for details). We run the same ML application, *Matrix Factorization* [25] (Section 5.2), on both systems.

For each system, we evaluate both BSP and SSP as the synchronization model (Section 2.1), with four deployment settings: (1) *LAN*, deployment within a single data center, (2) *EC2-ALL*, deployment across 11 aforementioned EC2 regions, (3) *V/C WAN*, deployment across two data centers that have the same WAN bandwidth as that between Virginia and California (Figure 2), representing a distributed ML setting within a continent, and (4) *S/S WAN*, deployment across two data centers that have the same WAN bandwidth as that between Singapore and São Paulo, representing the lowest WAN bandwidth between any two Amazon EC2 regions.

Figure 3 shows the normalized execution time until algorithm convergence across the four deployment settings. All results are normalized to IterStore using BSP on a LAN. The data label on each bar represents how much slower the WAN setting is than its *respective* LAN setting

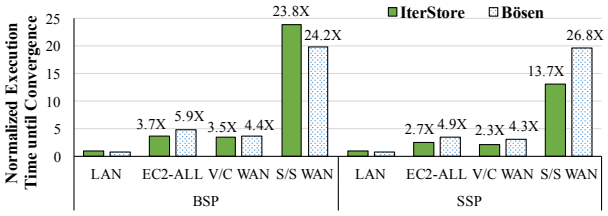


Figure 3: Normalized execution time until ML algorithm convergence when deploying two state-of-the-art distributed ML systems on a LAN and WANs

for the given system, e.g., Bösen-BSP on EC2-ALL is 5.9× slower than Bösen-BSP on LAN.

As we see, both systems suffer significant performance degradation when deployed across multiple data centers. When using BSP, IterStore is 3.5× to 23.8× slower on WANs than it is on a LAN, and Bösen is 4.4× to 24.2× slower. While using SSP can reduce overall execution times of both systems, both systems still show significant slowdown when run on WANs (2.3× to 13.7× for IterStore, and 4.3× to 26.8× for Bösen). We conclude that simply running state-of-the-art distributed ML systems on WANs can seriously slow down ML applications, and thus we need a new distributed ML system that can be effectively deployed on WANs.

3. Our Approach: Gaia

We introduce Gaia, a general ML system that can be effectively deployed on WANs to address the increasing need to run ML applications *directly* on geo-distributed data. We identify two key challenges in designing such a system (Section 3.1). We then introduce the system architecture of Gaia, which differentiates the communication *within* a data center from the communication *between* different centers (Section 3.2). Our approach is based on the key empirical finding that the vast majority of communication within an ML system results in *insignificant* changes to the state of the global model (Section 3.3). In light of this finding, we design a new ML synchronization model, called *Approximate Synchronous Parallel (ASP)*, which can eliminate the insignificant communication while ensuring the convergence and accuracy of ML algorithms. We describe ASP in detail in Section 3.4. Finally, Section 3.5 summarizes our theoretical analysis of how ASP guarantees algorithm convergence for a widely-used ML algorithm, stochastic gradient descent (SGD) (the full proof is in Appendix A).

3.1. Key Challenges

There are two key challenges in designing a general and effective ML system on WANs.

Challenge 1. *How to effectively communicate over WANs while retaining algorithm convergence and accuracy?* As we see above, state-of-the-art distributed ML systems can overwhelm the scarce WAN bandwidth, causing significant slowdowns. We need a mechanism that significantly reduces the communication between data centers so that the system can provide competitive performance. However, reducing communication can affect the accuracy of an ML algorithm. A poor choice

of synchronization model in a distributed ML systen prevent the ML algorithm from converging to the op point (i.e., the best model to explain or fit the input that one can achieve when using a proper synchroniz model [11, 59]. Thus, we need a mechanism tha reduce communication intensity while ensuring tha communication occurs in a *timely* manner, even whe network bandwidth is extremely stringent. This m nism should provably guarantee algorithm converg *irrespective* of the network conditions.

Challenge 2. *How to make the system generic and for ML algorithms without requiring modification?* D oping an effective ML algorithm takes significant e and experience, making it a large burden for the ML rithm developers to change the algorithm when deplo it on WANs. Our system should work across a wide variety of ML algorithms, preferably *without any change* to the algorithms themselves. This is challenging because different ML algorithms have different communication patterns, and the implication of reducing communication can vary significantly among them [37, 54, 60, 64, 66, 69, 83].

3.2. Gaia System Overview

We propose a new ML system, Gaia, that addresses the two key challenges in designing a general and effective ML system on WANs. Gaia is built on top the popular parameter server architecture, which is proven to be effective on a wide variety of ML algorithms (e.g., [5, 6, 13, 16, 17, 20, 34, 45, 74, 77]). As discussed in Section 2.1, in the parameter server architecture, *all* worker machines synchronize with each other through parameter servers to ensure that the global model state is up-to-date. While this architecture guarantees algorithm convergence, it also requires substantial communication between worker machines and parameter servers. To make Gaia effective on WANs while fully utilizing the abundant LAN bandwidth, we design a new system architecture to *decouple* the synchronization within a data center (LANs) from the synchronization across different data centers (WANs).

Figure 4 shows an overview of Gaia. In Gaia, each data center has some worker machines and parameter servers. Each worker machine processes a shard of the input data stored in its data center to achieve data parallelism (Section 2.1). The parameter servers in each data center collectively maintain a version of the *global model copy* (❶), and each parameter server handles a shard of this global model copy. A worker machine *only* `READ`s and `UPDATE`s the global model copy in its data center.

To reduce the communication overhead over WANs, the global model copy in each data center is only *approximately correct*. This design enables us to eliminate the insignificant, and thus unnecessary, communication across different data centers. We design a new synchronization model, called Approximate Synchronous Parallel (ASP ❷), between parameter servers *across* different data centers to ensure that each global model copy is approximately correct even with very low WAN bandwidth.

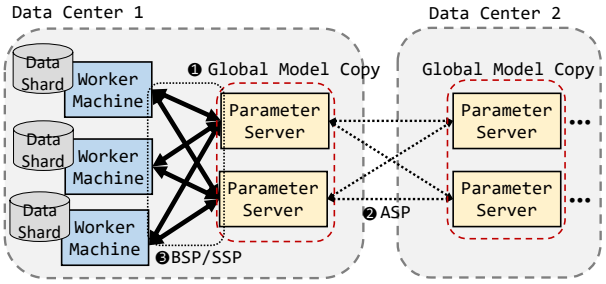


Figure 4: Gaia system overview

Section 3.4 describes the details of ASP. On the other hand, worker machines and parameter servers *within* a data center synchronize with each other using the conventional BSP (Bulk Synchronous Parallel) or SSP (Stale Synchronous Parallel) models (❸). These models allow worker machines to quickly observe fresh updates that happen *within* a data center. Furthermore, worker machines and parameter servers within a data center can employ more aggressive communication schemes such as sending updates early and often [19, 74] to fully utilize the abundant (and free) network bandwidth on a LAN.

3.3. Study of Update Significance

As discussed above, Gaia reduces the communication overhead over WANs by eliminating insignificant communication. To understand the benefit of our approach, we study the *significance* of the updates sent from worker machines to parameter servers. We study three classes of popular ML algorithms: *Matrix Factorization (MF)* [25], *Topic Modeling (TM)* [10], and *Image Classification (IC)* [43] (see Section 5.2 for descriptions). We run all the algorithms until convergence, analyze all the updates sent from worker machines to parameter servers, and compare the change they cause on the parameter value when the servers receive them. We define an update to be *significant* if it causes *S%* change on the parameter value, and we vary *S*, the significance threshold, between 0.01 and 10. Figure 5 shows the percentage of insignificant updates among all updates for different values of *S*.

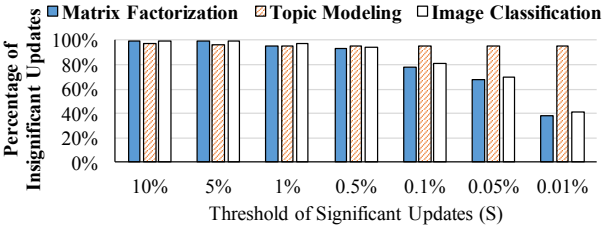


Figure 5: Percentage of insignificant updates

As we see, the vast majority of updates in these algorithms are *insignificant*. Assuming the significance threshold is 1%, 95.2% / 95.6% / 97.0% of all updates are insignificant for *MF* / *TM* / *IC*. When we relax the significance threshold to 5%, 98.8% / 96.1% / 99.3% of all updates are insignificant. Thus, most of the communication changes the ML model state only very slightly.

It is worth noting that our finding is consistent with the findings of prior work [21, 22, 40, 47, 80] on other

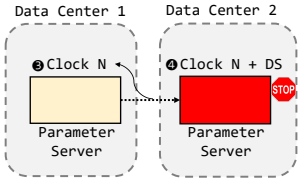
ML algorithms, such as PageRank and Lasso. These works observe that in these ML algorithms, not all model parameters converge to their optimal value within t same number iterations — a property called *non-unifo convergence* [78]. Instead of examining the converger rate, we *quantify* the *significance* of updates with v ious significance thresholds, which provides a uniq opportunity to reduce the communication over WANs

3.4. Approximate Synchronous Parallel

The goal of our new synchronization model, Appro mate Synchronous Parallel (ASP), is to ensure that t global model copy in each data center is approximat correct. In this model, a parameter server shares or the significant updates with other data centers, and A ensures that these updates can be seen by all data cent in a timely fashion. ASP achieves this goal by using thr techniques: (1) the significance filter, (2) ASP selecti barrier, and (3) ASP mirror clock. We describe them in order.

The significance filter. ASP takes two inputs from an ML programmer to determine whether or not an update is significant. They are: (1) a *significance function* and (2) an *initial significance threshold*. The significance function returns the significance of each update. We define an update as significant if its significance is larger than the threshold. For example, an ML programmer can define the significance function as the update's magnitude relative to the current value ($|\frac{Update}{Value}|$), and set the initial significance threshold to 1%. The significance function can be more sophisticated if the impact of parameter changes to the model is not linear, or the importance of parameters is non-uniform (see Section 4.3). A parameter server aggregates updates from the local worker machines and shares the aggregated updates with other data centers when the aggregated updates become significant. To ensure that the algorithm can converge to the optimal point, ASP automatically reduces the significance threshold over time (specifically, if the original threshold is v, then the threshold at iteration t of the ML algorithm is $v/\sqrt{t}$).

ASP selective barrier. While we can greatly reduce the communication overhead over WANs by sending only the significant updates, the WAN bandwidth might still be insufficient for such updates. In such a case, the significant updates can arrive too late, and we might not be able to bound the deviation between different global model copies. ASP handles this case with the *ASP selective barrier* (Figure 6a) control message. When a parameter server receives the significant updates (❶) at a rate that is higher than the WAN bandwidth can support, the parameter server first sends the indexes of these significant updates (as opposed to sending both the indexes and the update values together) via an ASP selective barrier (❷) to the other data centers. The receiver of an ASP selective barrier blocks its local worker from reading the specified parameters until it receives the significant updates from the sender of the barrier. This technique ensures that all worker machines in each data center are aware of the significant updates after a bounded network latency, and they wait *only* for these updates.



(b) *Mirror clock*

Figure 6: The synchronization mechanisms of ASP

Mirror clock. The ASP select barrier ensures that the latency of the significant updates is no more than the network latency. However, it assumes that 1) the underlying WAN bandwidth and latency are fixed so that the network latency can be bounded, and 2) such latency is short enough so that other data centers can be aware of them in time. In practice, WAN bandwidth can fluctuate over time [35], and the WAN latency can be intolerably high for some ML algorithms. We need a mechanism to guarantee that the worker machines are aware of the significant updates in time, irrespective of the WAN bandwidth or latency.

We use the *mirror clock* (Figure 6b) to provide this guarantee. When each parameter server receives all the updates from its local worker machines at the end of a clock (e.g., an iteration), it reports its clock to the servers that are in charge of the same parameters in the other data centers. When a server detects its clock is ahead of the slowest server that shares the same parameters by a predefined threshold DS (data center staleness), the server blocks its local worker machines from reading its parameters until the slowest mirror server catches up. In the example of Figure 6b, the server clock in Data Center 1 is N, while the server clock in Data Center 2 is $(N+DS)$. As their difference reaches the predefined limit, the server in Data Center 2 blocks its local worker from reading its parameters. This mechanism is similar to the concept of SSP [34], but we use it only as the last resort to guarantee algorithm convergence.

3.5. Summary of Convergence Proof

In this section, we summarize our proof showing that a popular, broad class of ML algorithms are guaranteed to converge under our new ASP synchronization model. The class we consider are ML algorithms expressed as convex optimization problems that are solved using distributed stochastic gradient descent.

The proof follows the outline of prior work on SSP [34], with a new challenge, i.e., our new ASP synchronization model allows the synchronization of insignificant updates to be delayed indefinitely. To prove algorithm convergence, our goal is to show that the distributed execution of an ML algorithm results in a set of parameter values that are very close (practically identical) to the values that would be obtained under a serialized execution.

Let f denote the objective function of an optimization problem, whose goal is to minimize f. Let $\tilde{\boldsymbol{x}}_t$ denote the sequence of noisy (i.e., inaccurate) *views* of the parameters, where $t = 1, 2, ..., T$ is the index of each view over time. Let $\boldsymbol{x}^*$ denote the value that minimizes f. Intuitively, we would like $f_t(\tilde{\boldsymbol{x}}_t)$ to approach $f(\boldsymbol{x}^*)$ as $t \to \infty$. We call the difference between $f_t(\tilde{\boldsymbol{x}}_t)$ and $f(\boldsymbol{x}^*)$ *regret*. We can prove $f_t(\tilde{\boldsymbol{x}}_t)$ approaches $f(\boldsymbol{x}^*)$ as $t \to \infty$ by proving that the *average regret*, $\frac{R[X]}{T} \to 0$ as $T \to \infty$.

Mathematically, the above intuition is formulated with Theorem 1. The details of the proof and the notations are in Appendix A.

Theorem 1. ***(Convergence of SGD under ASP)****. Suppose that, in order to compute the minimizer x^* of a convex function $f(\boldsymbol{x}) = \sum_{t=1}^{T} f_t(\boldsymbol{x})$, with $f_t, t = 1, 2, \ldots, T$, convex, we use stochastic gradient descent on one component ∇f_t at a time. Suppose also that 1) the algorithm is distributed in D data centers, each of which uses P machines, 2) within each data center, the SSP protocol is used, with a fixed staleness of s, and 3) a fixed mirror clock difference Δ_c is allowed between any two data centers. Let $\boldsymbol{u}_t = -\eta_t \nabla f_t(\tilde{\boldsymbol{x}}_t)$, where the step size η_t decreases as $\eta_t = \frac{\eta}{\sqrt{t}}$ and the significance threshold v_t decreases as $v_t = \frac{v}{\sqrt{t}}$. If we further assume that: $\|\nabla f_t(\boldsymbol{x})\| \leq L$, $\forall \boldsymbol{x} \in dom(f_t)$ and $\max(D(\boldsymbol{x}, \boldsymbol{x}')) \leq \Delta^2, \forall \boldsymbol{x}, \boldsymbol{x}' \in dom(f_t)$. Then, as $T \to \infty$, the regret $R[X] = \sum_{t=1}^{T} f_t(\tilde{\boldsymbol{x}}_t) - f(\boldsymbol{x}^*) = O(\sqrt{T})$ and therefore $\lim_{T\to\infty} \frac{R[X]}{T} \to 0$.*

4. Implementation

We introduce the key components of Gaia in Section 4.1, and discuss the operation and design of individual components in the remaining sections.

4.1. Gaia System Key Components

Figure 7 presents the key components of Gaia. All of the key components are implemented in the parameter servers, and can be transparent to the ML programs and the worker machines. As we discuss above, we decouple the synchronization within a data center (LANs) from the synchronization across different data centers (WANs). The *local server* (❶) in each parameter server handles the synchronization between the worker machines in the same data center using the conventional BSP or SSP models. On the other hand, the *mirror server* (❷) and the *mirror client* (❸) handle the synchronization with other

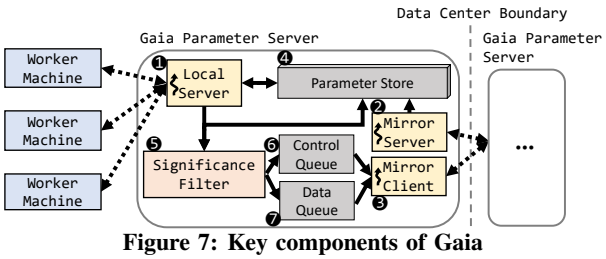


Figure 7: Key components of Gaia

4.2. System Operations and Communication

We present a walkthrough of major system operations and communication.

`UPDATE` from a worker machine. When a *local server* (❶) receives a parameter update from a worker machine, it updates the parameter in its *parameter store* (❹), which maintains the parameter value and its accumulated update. The local server then invokes the *significance filter* (❺) to determine whether or not the accumulated update of this parameter is significant. If it is, the significance filter sends a `MIRROR UPDATE` request to the mirror client (❸) and resets the accumulated update for this parameter.

Messages from the significance filter. The significance filter sends out three types of messages. First, as discussed above, it sends a `MIRROR UPDATE` request to the mirror client through the data queue (❼). Second, when the significance filter detects that the arrival rate of significant updates is higher than the underlying WAN bandwidth that it monitors at every iteration, it first sends an `ASP Barrier` (Section 3.4) to the control queue (❻) before sending the `MIRROR UPDATE`. The mirror client (❸) prioritizes the control queue over the data queue, so that the barrier is sent out earlier than the update. Third, to maintain the mirror clock (Section 3.4), the significance filter also sends a `MIRROR CLOCK` request to the control queue at the end of each clock in the local server.

Operations in the mirror client. The mirror client thread wakes up when there is a request from the control queue or the data queue. Upon waking up, the mirror client walks through the queues, packs together the messages to the same destination, and sends them.

Operations in the mirror server. The mirror server handles above messages (`MIRROR UPDATE`, `ASP BARRIER`, and `MIRROR CLOCK`) according to our ASP model. For `MIRROR UPDATE`, it applies the update to the corresponding parameter in the parameter store. For `ASP BARRIER`, it sets a flag in the parameter store to block the corresponding parameter from being read until it receives the corresponding `MIRROR UPDATE`. For `MIRROR CLOCK`, the mirror server updates its local mirror clock state for each parameter server in other data centers, and enforces the predefined clock difference threshold *DS* (Section 3.4).

4.3. Advanced Significance Functions

As we discuss in Section 3.4, the significance filter allows the ML programmer to specify a custom *significance function* to calculate the significance of each update. By providing an advanced significance function, Gaia can be more effective at eliminating the insignificant communication. If several parameters are always referenced together to calculate the next update, the significance function can take into account the values of all these parameters. For example, if three parameters a, b, and c are always used as $a \cdot b \cdot c$ in an ML algorithm, the significance of a, b, and c can be calculated as the change on $a \cdot b \cdot c$. If one of them is 0, any change in another parameter, however large it may be, is insignificant. Similar principles can be applied to model parameters that are non-linear or

non-uniform. For unmodified ML programs, the system applies default significance functions, such as the relative magnitude of an update for each parameter.

4.4. Tuning of Significance Thresholds

The user of Gaia can specify two different goals for Gaia: (1) speed up algorithm convergence by fully utilizing the available WAN bandwidth and (2) minimize the communication cost on WANs. In order to achieve either of these goals, the significance filter maintains two significance thresholds and dynamically tunes these thresholds. The first threshold is the *hard* significance threshold. The purpose of this threshold is to guarantee ML algorithm convergence. As we discuss in our theoretical analysis (Section 3.5), the initial threshold is provided by the ML programmer or a default system setting, and the significance filter reduces it over time. Every update whose significance is above the hard threshold is guaranteed to be sent to other data centers. The second threshold is the *soft* significance threshold. The purpose of it is to use underutilized WAN bandwidth to speed up convergence. This threshold is tuned based on the arrival rate of the significant updates and the underlying WAN bandwidth. When the user chooses to optimize the first goal (speed up algorithm convergence), the system lowers the soft significance threshold whenever there is underutilized WAN bandwidth. The updates whose significance is larger than the soft significance threshold are sent in a best-effort manner. On the other hand, if the goal of the system is to minimize the WAN communication costs, the soft significance threshold is not activated.

While the configuration of the initial hard threshold depends on how error tolerant each ML algorithm is, a simple and conservative threshold (such as 1%–2%) is likely to work in most cases. This is because most ML algorithms initialize their parameters with random values, and make large changes to their model parameters at early phases. Thus, they are more error tolerant at the beginning. As Gaia reduces the threshold over time, its accuracy loss is limited. An ML expert can choose a more aggressive threshold based on domain knowledge of the ML algorithm.

4.5. Overlay Network and Hub

While Gaia can eliminate the insignificant updates, each data center needs to *broadcast* the significant updates to all the other data centers. This broadcast-based communication could limit the scalability of Gaia when we deploy Gaia to many data centers. To make Gaia more scalable with more data centers, we use the concept of overlay networks [48].

As we discuss in Section 2.2, the WAN bandwidth between geographically-close regions is much higher than that between distant regions. In light of this, Gaia supports having geographically-close data centers form a *data center group*. Servers in a data center group send their significant updates only to the other servers in the same group. Each group has *hub* data centers that are in charge of aggregating all the significant updates within the group, and sending to the hubs of the other groups. Similarly, a hub data center broadcasts the aggregated significant updates from other groups to the other data centers within its group. Each data center group can designate different hubs for communication with different data center groups, so the system can utilize more links within a data center group. For example, the data centers in Virginia, California, and Oregon can form a data center group and assign the data center in Virginia as the hub to communicate with the data centers in Europe and the data center in Oregon as the hub to communicate with the data centers is Asia. This design allows Gaia to broadcast the significant updates with lower communication cost.

5. Methodology

5.1. Experiment Platforms

We use three different platforms for our evaluation.

Amazon-EC2. We deploy Gaia to 22 machines spread across 11 EC2 regions as we show in Figure 2. In each EC2 region we start two instances of type `c4.4xlarge` or `m4.4xlarge` [8], depending on their availability. Both types of instances have 16 CPU cores and at least 30GB RAM, running 64-bit Ubuntu 14.04 LTS (HVM). In all, our deployment uses 352 CPU cores and 1204 GB RAM.

Emulation-EC2. As the monetary cost of running all experiments on EC2 is too high, we run some experiments on our local cluster that emulates the computation power and WAN bandwidth of EC2. We use the same number of machines (22) in our local cluster. Each machine is equipped with a 16-core Intel Xeon CPU (E5-2698), an NVIDIA Titan X GPU, 64GB RAM, a 40GbE NIC, and runs the same OS as above. The computation power and the LAN speeds of our machines are higher than the ones we get from EC2, so we slow down the CPU and LAN speeds to match the speeds on EC2. We model the measured EC2 WAN bandwidth (Figure 2) with the Linux Traffic Control tool [3]. As Section 6.1 shows, our emulation platform gives very similar results to the results from our real EC2 deployment.

Emulation-Full-Speed. We run some of our experiments on our local cluster that emulates the WAN bandwidth of EC2 at *full speed*. We use the same settings as **Emulation-EC2** except we do not slow down the CPUs and the LAN. We use this platform to show the results of deployments with more powerful nodes.

5.2. Applications

We evaluate Gaia with three popular ML applications.

Matrix Factorization (MF) is a technique commonly used in recommender systems, e.g., systems that recommend movies to users on Netflix (a.k.a. collaborative filtering) [25]. Its goal is to discover latent interactions between two entities, such as users and movies, via matrix factorization. For example, input data can be a partially filled matrix X, where every entry is a user's rating for a movie, each row corresponding to a user, and each column corresponding to a specific movie. Matrix factor-

ization factorizes X into factor matrices L and R such that their product approximates X (i.e., $X \approx LR$). Like other systems [17, 32, 83], we implement *MF* using the stochastic gradient descent (SGD) algorithm. Each worker is assigned a portion of the known entries in X. The L matrix is stored locally in each worker, and the R matrix is stored in parameter servers. Our experiments use the *Netflix* dataset, a 480K-by-18K sparse matrix with 100M known entries. They are configured to factor the matrix into the product of two matrices, each with rank 500.

Topic Modeling (TM) is an unsupervised method for discovering hidden semantic structures (*topics*) in an unstructured collection of *documents*, each consisting of a bag (multi-set) of *words* [10]. *TM* discovers the topics via word co-occurrence. For example, "policy" is more likely to co-occur with "government" than "bacteria", and thus "policy" and "government" are categorized to the same topic associated with political terms. Further, a document with many instances of "policy" would be assigned a topic distribution that peaks for the politics-related topics. *TM* learns the hidden topics and the documents' associations with those topics jointly. Common applications for *TM* include community detection in social networks and news categorizations. We implement our *TM* solver using collapsed Gibbs sampling [30]. We use the *Nytimes* dataset [53], which has 100M words in 300K documents with a vocabulary size of 100K. Our experiments classify words and documents into 500 topics.

Image Classification (IC) is a task to classify images into categories, and the state-of-the-art approach is using deep learning and convolutional neural networks (CNNs) [43]. Given a set of images with known categories (training data), the ML algorithm trains a CNN to learn the relationship between the image features and their categories. The trained CNN is then used to predict the categories of another set of images (test data). We use GoogLeNet [65], one of the state-of-the-art CNNs as our model. We train GoogLeNet using stochastic gradient descent with back propagation [61]. As training a CNN with a large number of images requires substantial computation, doing so on CPUs can take hundreds of machines over a week [13]. Instead, we use distributed GPUs with a popular deep learning framework, Caffe [38], which is hosted by a state-of-the-art GPU-specialized parameter server system, GeePS [18]. Our experiments use the ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC12) [62] dataset, which consists of 1.3M training images and 50K test images. Each image is labeled as one of the 1,000 pre-defined categories.

5.3. Performance Metrics and Algorithm Convergence Criteria

We use two performance metrics to evaluate the effectiveness of a globally distributed ML system. The first metric is the *execution time until algorithm convergence*. We use the following algorithm convergence criterion, based on guidance from our ML experts: if the value of the objective function (the *objective value*) in an algorithm changes by less than 2% over the course of 10 iterations, we declare that the algorithm has converged [32]. In order to ensure that each algorithm *accurately* converges to the optimal point, we first run each algorithm on our local cluster until it converges, and we record the absolute objective value. The execution time of each setting is the time it takes to converge to this absolute objective value. The second metric is the *cost of algorithm convergence*. We calculate the cost based on the cost model of Amazon EC2 [8], including the cost of the server time and the cost of data transfer on WANs. We provide the details of the cost model in Appendix C.

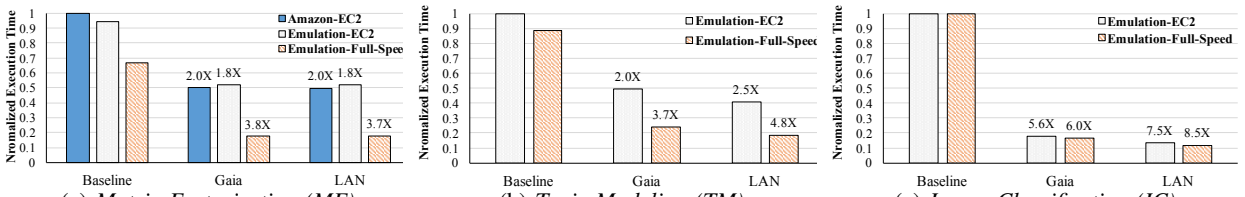
6. Evaluation Results

We evaluate the effectiveness of `Gaia` by evaluating three types of systems/deployments: (1) `Baseline`, two state-of-the-art parameter server systems (IterStore [17] for *MF* and *TM*, GeePS [18] for *IC*) that are deployed across multiple data centers. Every worker machine handles the data in its data center, while the parameter servers are distributed evenly across all the data centers; (2) `Gaia`, our prototype systems based on IterStore and GeePS, deployed across multiple data centers; and (3) `LAN`, the baseline parameter servers (IterStore and GeePS) that are deployed within a single data center (also on 22 machines) that already hold all the data, representing the ideal case of all communication on a LAN. For each system, we evaluate two ML synchronization models: *BSP* and *SSP* (Section 2.1). For `Baseline` and `LAN`, BSP and SSP are used among all worker machines, whereas for `Gaia`, they are used only within each data center. Due to limited space, we present the results for BSP in this section and leave the results for SSP to Appendix B.

6.1. Performance on EC2 Deployment

We first present the performance of `Gaia` and `Baseline` when they are deployed across 11 EC2 data centers. Figure 8 shows the normalized execution time until convergence for our ML applications, normalized to `Baseline` on EC2. The data label on each bar is the speedup over `Baseline` for *the respective deployment*. As Section 5.1 discusses, we run only *MF* on EC2 due to the high monetary cost of WAN data transfer. Thus, we present the results of *MF* on all three platforms, while we show the results of *TM* and *IC* only on our emulation platforms. As Figure 8a shows, our emulation platform (*Emulation-EC2*) matches the execution time of our real EC2 deployment (*Amazon-EC2*) very well. We make two major observations.

First, we find that `Gaia` significantly improves the performance of `Baseline` when deployed globally across many EC2 data centers. For *MF*, `Gaia` provides a speedup of 2.0× over `Baseline`. Furthermore, the performance of `Gaia` is very similar to the performance of `LAN`, indicating that `Gaia` almost attains the performance *upper bound* with the given computation resources. For *TM*, `Gaia` delivers a similar speedup (2.0×) and is within 1.25× of the ideal speed of `LAN`. For *IC*, `Gaia` provides a speedup



(a) *Matrix Factorization (MF)* (b) *Topic Modeling (TM)* (c) *Image Classification (IC)*

Figure 8: Normalized execution time until convergence when deployed across 11 EC2 regions and our emulation cluster

of 5.6× over `Baseline`, which is within 1.32× of the `LAN` speed, indicating that `Gaia` is also effective on a GPU-based ML system. The gap between `Baseline` and `LAN` is larger for *IC* than for the other two applications. This is because the GPU-based ML system generates parameter updates at a higher rate than the CPU-based one, and therefore the limited WAN bandwidth slows it down more significantly.

Second, `Gaia` provides a higher performance gain when deployed on a more powerful platform. As Figure 8 shows, the performance gap between `Baseline` and `LAN` significantly increases on *Emulation-Full-Speed* compared to the slower platform *Emulation-EC2*. This is expected because the WAN bandwidth becomes a more critical bottleneck when the computation time reduces and the LAN bandwidth increases. `Gaia` successfully mitigates the WAN bottleneck in this more challenging *Emulation-Full-Speed* setting, and improves the system performance by 3.8× for *MF*, 3.7× for *TM*, and 6.0× for *IC* over `Baseline`, approaching the speedups provided by `LAN`.

6.2. Performance and WAN Bandwidth

To understand how `Gaia` performs under different amounts of WAN bandwidth, we evaluate two settings where `Baseline` and `Gaia` are deployed across two data centers with two WAN bandwidth configurations: (1) *V/C WAN*, which emulates the WAN bandwidth between Virginia and California, representing a setting within the same continent; and (2) *S/S WAN*, which emulates the WAN bandwidth between Singapore and São Paulo, representing the lowest WAN bandwidth between any two Amazon EC2 sites. All the experiments are conducted on our emulation platform at full speed. Figures 9 and 10 show the results. Three observations are in order.

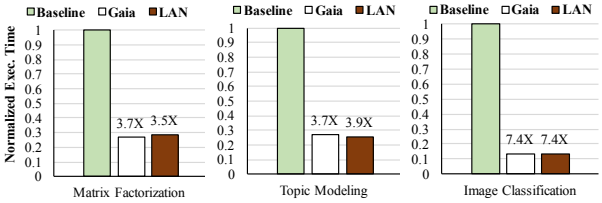


Figure 9: Normalized execution time until convergence with the WAN bandwidth between Virginia and California

First, `Gaia` successfully matches the performance of `LAN` when WAN bandwidth is high (*V/C WAN*). As Figure 9 shows, `Gaia` achieves a speedup of 3.7× for *MF*, 3.7× for *TM*, and 7.4× for *IC*. For all three ML applications, the performance of `Gaia` on WANs is almost the same as `LAN` performance.

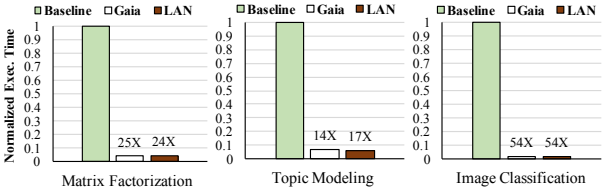


Figure 10: Normalized execution time until convergence with the WAN bandwidth between Singapore and São Paulo

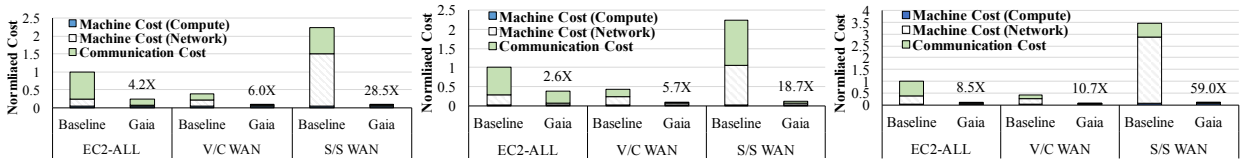
Second, `Gaia` still performs very well when WAN bandwidth is low (*S/S WAN*, Figure 10): `Gaia` provides a speedup of 25.4× for *MF*, 14.1× for *TM*, and 53.5× for *IC*, and successfully approaches `LAN` performance. These results show that our design is robust for both CPU-based and GPU-based ML systems, and it can deliver high performance even under scarce WAN bandwidth.

Third, for *MF*, the performance of `Gaia` (on WANs) is slightly better than `LAN` performance. This is because we run ASP between different data centers, and the workers in each data center need to synchronize only with each other locally in each iteration. As long as the mirror updates on WANs are timely, each iteration of `Gaia` can be faster than that of `LAN`, which needs to synchronize across all workers. While `Gaia` needs more iterations than `LAN` due to the accuracy loss, `Gaia` can still outperform `LAN` due to the faster iterations.

6.3. Cost Analysis

Figure 11 shows the monetary cost of running ML applications until convergence based on the Amazon EC2 cost model, normalized to the cost of `Baseline` on 11 EC2 regions. Cost is divided into three components: (1) the cost of machine time spent on computation, (2) the cost of machine time spent on waiting for networks, and (3) the cost of data transfer across different data centers. As we discuss in Section 2.2, there is no cost for data transfer within a single data center in Amazon EC2. The data label on each bar shows the factor by which the cost of `Gaia` is cheaper than the cost of *each respective* `Baseline`. We evaluate all three deployment setups that we discuss in Sections 6.1 and 6.2. We make two major observations.

First, `Gaia` is very effective in reducing the cost of running a geo-distributed ML application. Across all the evaluated settings, `Gaia` is 2.6× to 59.0× cheaper than `Baseline`. Not surprisingly, the major cost saving comes from the reduction of data transfer on WANs and the reduction of machine time spent on waiting for networks. For the S/S WAN setting, the cost of waiting for networks is a more important factor than the other



(a) *Matrix Factorization (MF)* (b) *Topic Modeling (TM)* (c) *Image Classification (IC)*

Figure 11: Normalized monetary cost of `Gaia` vs. `Baseline`

two settings, because it takes more time to transfer the same amount of data under low WAN bandwidth. As `Gaia` significantly improves system performance and reduces data communication overhead, it significantly reduces both cost sources. We conclude that `Gaia` is a cost-effective system for geo-distributed ML applications.

Second, `Gaia` reduces data transfer cost much more when deployed on a smaller number of data centers. The reason is that `Gaia` needs to broadcast the significant updates to *all* data centers, so communication cost is higher as the number of data centers increases. While we employ network overlays (Section 4.5) to mitigate this effect, there is still more overhead with more than two data centers. Nonetheless, the cost of `Gaia` is still much cheaper (4.2×/2.6×/8.5×) than `Baseline` even when deployed across 11 data centers.

6.4. Comparisons with Centralized Data

Gaia obtains its good performance without moving any raw data, greatly reducing WAN costs and respecting privacy and data sovereignty laws that *prohibit* raw data movement. For settings in which raw data movement is *allowed*, Table 1 summarizes the performance and cost comparisons between `Gaia` and the centralized data approach (`Centralized`), which moves *all* the geo-distributed data into a single data center and then runs the ML application over the data. We make `Centralized` very cost efficient by moving the data into the *cheapest* data center in each setting, and we use low cost machines (`m4.xlarge` [8]) to move the data. We make two major observations.

Table 1: Comparison between `Gaia` and `Centralized`

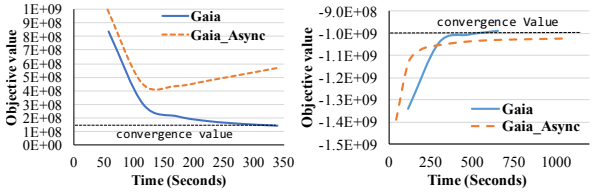
Application	Setting	`Gaia` Speedup over `Centralized`	`Gaia` cost / `Centralized` cost
MF	EC2-ALL	1.11	3.54
	V/C WAN	1.22	1.00
	S/S WAN	2.13	1.17
TM	EC2-ALL	0.80	6.14
	V/C WAN	1.02	1.26
	S/S WAN	1.25	1.92
IC	EC2-ALL	0.76	3.33
	V/C WAN	1.12	1.07
	S/S WAN	1.86	1.08

First, `Gaia` outperforms `Centralized` for most settings, except for *TM* and *IC* in the EC2-ALL setting. Other than these two cases, `Gaia` provides a 1.02–2.13× speedup over `Centralized`. This is because `Gaia` does not need to wait for data movement over WANs, and the performance of `Gaia` is very close to that of `LAN`. On the other hand, `Centralized` performs better when there is a performance gap between `Gaia` and `LAN`, especially in the setting of all 11 data centers for *TM* and *IC*. The data movement overhead of `Centralized` is smaller in this setting because each data center has only a small fraction of the data, and `Centralized` moves the data from all data centers in parallel.

Second, `Centralized` is more cost-efficient than `Gaia`, but the gap is small in the two data centers setting. This is because the total WAN traffic of `Gaia` is still larger than the size of the training data, even though `Gaia` significantly reduces the communication overhead over `Baseline`. The cost gap is larger in the setting of 11 data centers (3.33–6.14×) than in two data centers (1.00–1.92×), because the WAN traffic of `Gaia` is positively correlated with the number of data centers (Section 4.5).

6.5. Effect of Synchronization Mechanisms

One of the major design considerations of ASP is to ensure that the significant updates arrive in a timely manner to guarantee algorithm convergence. To understand the effectiveness of our proposed synchronization mechanisms (i.e., ASP selective barrier and mirror clock), we run *MF* and *TM* on `Gaia` with both mechanisms disabled across 11 EC2 regions. Figure 12 shows the progress toward algorithm convergence with the synchronization mechanisms enabled (`Gaia`) and disabled (`Gaia_Async`). For *MF*, lower object value is better, while for *TM*, higher is better.



(a) *Matrix Factorization (MF)* (b) *Topic Modeling (TM)*

Figure 12: Progress toward algorithm convergence with and without Gaia's synchronization mechanisms

As Figure 12 shows, `Gaia` steadily reaches algorithm convergence for both applications. In contrast, `Gaia_Async` diverges from the optimum point at ~100 seconds for *MF*. For *TM*, `Gaia_Async` looks like it makes faster progress at the beginning of execution because it eliminates the synchronization overhead. However, it makes very slow progress after ~200 seconds and does not reach the value that results in convergence until 1100 seconds. It may take a long time for `Gaia_Async` to reach that point, if ever. Thus, the lack of synchronization leads to worse model quality than that achieved by using proper synchronization mechanisms. Both results demonstrate that the synchronization mechanisms we introduce in ASP are effective and vital for deploying ML algorithms on Gaia on WANs.

7. Related Work

To our knowledge, this is the first work to propose a globally distributed ML system that is (1) designed to run effectively over WANs while ensuring the convergence and accuracy of an ML algorithm, and (2) generic and flexible to run a wide range of ML algorithms *without* requiring modification. In this section, we discuss related works on 1) WAN-aware data analytics and ML systems, 2) distributed ML systems, 3) non-uniform convergence in ML algorithms, 4) communication-efficient ML algorithms, and 5) approximate queries on distributed data, all of which are related to our proposed system.

WAN-aware Data Analytics and ML Systems. Prior work establishes the emerging problem of analyzing the globally-generated data in the context of data analytics systems (e.g., [33, 36, 41, 57, 58, 71–73]). These works show very promising WAN bandwidth reduction and/or system performance improvement with a WAN-aware data analytics framework. However, their goal is *not* to run an *ML system* effectively on WANs, which has very different challenges from a data analytics system. Cano et al. [12] first discuss the problem of running an ML system on geo-distributed data. They show that a geo-distributed ML system can perform significantly better by leveraging a communication-efficient algorithm [49] for logistic regression models. This solution requires the ML programmer to change the ML algorithm, which is challenging and algorithm-specific. In contrast, our work focuses on the design of communication-efficient mechanisms at the system level and requires no modification to the ML algorithms.

Distributed ML Systems. There are many distributed ML systems that aim to enable large-scale ML applications (e.g., [1, 2, 5, 6, 13, 16–18, 20, 34, 40, 45–47, 74, 77]). These systems successfully demonstrate their effectiveness on a large number of machines by employing various synchronization models and system optimizations. However, all of them assume that the network communication happens *within* a data center and do not tackle the challenges of scarce WAN bandwidth. As our study shows, state-of-the-art parameter server systems suffer from significant performance degradation and data transfer cost when deployed across multiple data centers on WANs. We demonstrate our idea on both CPU-based and GPU-based parameter server systems [17, 18], and we believe our proposed general solution can be applied to all distributed ML systems that use the parameter server architecture.

Non-uniform Convergence in ML Algorithms. Prior work observes that, in many ML algorithms, not all model parameters converge to their optimal value within the same number of computation iterations [21, 22, 40, 47, 80]. Several systems exploit this property to improve the algorithm convergence speed, e.g., by prioritizing the computation of important parameters [40, 47, 80], communicating the important parameters more aggressively [74], or sending fewer updates with user-defined filters [45, 46].

Among these, the closest to our work is Li et al.'s proposal for a communication-efficient parameter server system [45, 46], which employs various filters to reduce communication between worker machines and parameter servers. In contrast to our work, this work 1) does not differentiate the communication on LANs from the communication on WANs, and thus it cannot make efficient use of the abundant LAN bandwidth or the scarce and heterogeneous WAN bandwidth, and 2) does not propose a general synchronization model across multiple data centers, a key contribution of our work.

Communication-Efficient ML Algorithms. A large body of prior work proposes ML algorithms to reduce the dependency on intensive parameter updates to enable more efficient parallel computation (e.g., [37, 52, 63, 66, 81, 82, 84]). These works are largely orthogonal to our work, as we focus on a generic system-level solution that does not require any changes to ML algorithms. These ML algorithms can use our system to further reduce communication overhead over WANs.

Approximate Queries on Distributed Data. In the database community, there is substantial prior work that explores the trade-off between precision and performance for queries on distributed data over WANs (e.g., [15, 55, 56, 75]). The idea is to reduce communication overhead of moving up-to-date data from data sources to a query processor by allowing some user-defined precision loss for the queries. At a high level, our work bears some resemblance to this idea, but the challenges we tackle are fundamentally different. The focus of distributed database systems is on approximating queries (e.g., simple aggregation) on raw data. In contrast, our work focuses on retaining ML algorithm convergence and accuracy by using approximate ML models during training.

8. Conclusion

We introduce Gaia, a new ML system that is designed to efficiently run ML algorithms on globally-generated data over WANs, without any need to change the ML algorithms. Gaia decouples the synchronization within a data center (LANs) from the synchronization across different data centers (WANs), enabling flexible and effective synchronization over LANs and WANs. We introduce a new synchronization model, Approximate Synchronous Parallel (ASP), to efficiently utilize the scarce and heterogeneous WAN bandwidth while ensuring convergence of the ML algorithms with a theoretical guarantee. Using ASP, Gaia dynamically eliminates insignificant, and thus unnecessary, communication over WANs. Our evaluation shows that Gaia significantly outperforms two state-of-the-art parameter server systems on WANs, and is within 0.94–1.40× of the speed of running the same ML algorithm on a LAN. Gaia also significantly reduces the monetary cost of running the same ML algorithm on WANs, by 2.6–59.0×. We conclude that Gaia is a practical and effective system to enable globally-distributed ML applications, and we believe the ideas behind Gaia's system design for communication across WANs can be applied to many other large-scale distributed ML systems.

Acknowledgments

We thank our shepherd, Adam Wierman, and the reviewers for their valuable suggestions. We thank the SAFARI group members and Garth A. Gibson for their feedback. Special thanks to Henggang Cui for his help on IterStore and GeePS, Jinliang Wei for his help on Bösen, and their feedback. We thank the members and companies of the PDL Consortium (including Broadcom, Citadel, EMC, Facebook, Google, HP Labs, Hitachi, Intel, Microsoft Research, MongoDB, NetApp, Oracle, Samsung, Seagate, Tintri, Two Sigma, Uber, Veritas, and Western Digital) for their interest, insights, feedback, and support. We acknowledge the support of our industrial partners: Google, Intel, NVIDIA, Samsung and VMWare. This work is supported in part by NSF grant 1409723, Intel STC on Cloud Computing (ISTC-CC), Intel STC on Visual Cloud Systems (ISTC-VCS), and the Dept of Defense under contract FA8721-05-C-0003. Dimitris Konomis is partially supported by Onassis Foundation.

References

[1] "Apache Mahout." http://mahout.apache.org/

[2] "Apache Spark MLlib." http://spark.apache.org/mllib/

[3] "Linux Traffic Control." http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html

[4] "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Process. Mag.*, 2012.

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. http://tensorflow.org/

[6] A. Ahmed, M. Aly, J. Gonzalez, S. M. Narayanamurthy, and A. J. Smola, "Scalable inference in latent variable models," in *WSDM*, 2012.

[7] Amazon, "AWS global infrastructure." https://aws.amazon.com/about-aws/global-infrastructure/

[8] Amazon, "Amazon EC2 pricing," Janurary 2017. https://aws.amazon.com/ec2/pricing/

[9] A. Auradkar, C. Botev, S. Das, D. D. Maagd, A. Feinberg, P. Ganti, L. Gao, B. Ghosh, K. Gopalakrishna, B. Harris, J. Koshy, K. Krawez, J. Kreps, S. Lu, S. Nagaraj, N. Narkhede, S. Pachev, I. Perisic, L. Qiao, T. Quiggle, J. Rao, B. Schulman, A. Sebastian, O. Seeliger, A. Silberstein, B. Shkolnik, C. Soman, R. Sumbaly, K. Surlaker, S. Topiwala, C. Tran, B. Varadarajan, J. Westerman, Z. White, D. Zhang, and J. Zhang, "Data infrastructure at LinkedIn," in *ICDE*, 2012.

[10] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet allocation," *JMLR*, 2003.

[11] J. K. Bradley, A. Kyrola, D. Bickson, and C. Guestrin, "Parallel coordinate descent for L1-regularized loss minimization," in *ICML*, 2011.

[12] I. Cano, M. Weimer, D. Mahajan, C. Curino, and G. M. Fumarola, "Towards geo-distributed machine learning," *CoRR*, 2016.

[13] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an efficient and scalable deep learning training system," in *OSDI*, 2014.

[14] C. Chu, S. K. Kim, Y. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun, "Map-Reduce for machine learning on multicore," in *NIPS*, 2006.

[15] G. Cormode, M. N. Garofalakis, S. Muthukrishnan, and R. Rastogi, "Holistic aggregates in a networked world: Distributed tracking of approximate quantiles," in *SIGMOD*, 2005.

[16] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Exploiting bounded staleness to speed up big data analytics," in *USENIX ATC*, 2014.

[17] H. Cui, A. Tumanov, J. Wei, L. Xu, W. Dai, J. Haber-Kucharsky, Q. Ho, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Exploiting iterative-ness for parallel ML computations," in *SoCC*, 2014, software available at https://github.com/cuihenggang/iterstore.

[18] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server," in *EuroSys*, 2016, software available at https://github.com/cuihenggang/geeps.

[19] W. Dai, A. Kumar, J. Wei, Q. Ho, G. Gibson, and E. P. Xing, "Analysis of high-performance distributed ML at scale through parameter server consistency models," in *AAAI*, 2015.

[20] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *NIPS*, 2012.

[21] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani, "Least angle regression," in *The Annals of Statistics*, 2004.

[22] G. Elidan, I. McGraw, and D. Koller, "Residual belief propagation: Informed scheduling for asynchronous message passing," in *UAI*, 2006.

[23] ESnet and Lawrence Berkeley National Laboratory, "iperf3." http://software.es.net/iperf/

[24] A. Frome, G. S. Corrado, J. Shlens, S. Bengio, J. Dean, M. Ranzato, and T. Mikolov, "DeViSE:

A deep visual-semantic embedding model," in *NIPS*, 2013.

[25] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *SIGKDD*, 2011.

[26] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *OSDI*, 2012.

[27] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *OSDI*, 2014.

[28] Google, "Google data center locations." https://www.google.com/about/datacenters/inside/locations/index.html

[29] A. G. Greenberg, J. R. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *Computer Communication Review*, 2009.

[30] T. L. Griffiths and M. Steyvers, "Finding scientific topics," *Proceedings of the National Academy of Sciences of the United States of America*, 2004.

[31] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal, "Mesa: Geo-replicated, near real-time, scalable data warehousing," *PVLDB*, 2014.

[32] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Addressing the straggler problem for iterative convergent parallel ML," in *SoCC*, 2016.

[33] B. Heintz, A. Chandra, and R. K. Sitaraman, "Optimizing grouped aggregation in geo-distributed streaming analytics," in *HPDC*, 2015.

[34] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing, "More effective distributed ML via a stale synchronous parallel parameter server," in *NIPS*, 2013.

[35] C. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *SIGCOMM*, 2013.

[36] C. Hung, L. Golubchik, and M. Yu, "Scheduling jobs across geo-distributed datacenters," in *SoCC*, 2015.

[37] M. Jaggi, V. Smith, M. Takác, J. Terhorst, S. Krishnan, T. Hofmann, and M. I. Jordan, "Communication-efficient distributed dual coordinate ascent," in *NIPS*, 2014.

[38] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *CoRR*, 2014.

[39] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and F. F. Li, "Large-scale video classification with convolutional neural networks," in *CVPR*, 2014.

[40] J. K. Kim, Q. Ho, S. Lee, X. Zheng, W. Dai, G. A. Gibson, and E. P. Xing, "STRADS: a distributed framework for scheduled model parallel machine learning," in *EuroSys*, 2016.

[41] K. Kloudas, R. Rodrigues, N. M. Preguiça, and M. Mamede, "PIXIDA: optimizing data parallel jobs in wide-area data analytics," *PVLDB*, 2015.

[42] N. Laoutaris, M. Sirivianos, X. Yang, and P. Rodriguez, "Inter-datacenter bulk transfers with netstitcher," in *SIGCOMM*, 2011.

[43] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, 1989.

[44] G. Lee, J. J. Lin, C. Liu, A. Lorek, and D. V. Ryaboy, "The unified logging infrastructure for data analytics at Twitter," *PVLDB*, 2012.

[45] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Su, "Scaling distributed machine learning with the parameter server," in *OSDI*, 2014.

[46] M. Li, D. G. Andersen, A. J. Smola, and K. Yu, "Communication efficient distributed machine learning with the parameter server," in *NIPS*, 2014.

[47] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning in the cloud," *VLDB*, 2012.

[48] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes," *IEEE Communications Surveys and Tutorials*, 2005.

[49] D. Mahajan, S. S. Keerthi, S. Sundararajan, and L. Bottou, "A functional approximation based distributed learning algorithm," *CoRR*, 2013.

[50] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "MLlib: Machine learning in Apache Spark," *CoRR*, 2015.

[51] Microsoft, "Azure regions." https://azure.microsoft.com/en-us/region

[52] W. Neiswanger, C. Wang, and E. P. Xing, "Asymptotically exact, embarrassingly parallel MCMC," in *UAI*, 2014.

[53] "New York Times dataset," http://www.ldc.upenn.edu/.

[54] D. Newman, A. U. Asuncion, P. Smyth, and M. Welling, "Distributed algorithms for topic models," *JMLR*, 2009.

[55] C. Olston, J. Jiang, and J. Widom, "Adaptive filters for continuous queries over distributed data streams," in *SIGMOD*, 2003.

[56] C. Olston and J. Widom, "Offering a precision-performance tradeoff for aggregation queries over replicated data," in *VLDB*, 2000.

[57] Q. Pu, G. Ananthanarayanan, P. Bodík, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low latency geo-distributed data analytics," in *SIGCOMM*, 2015.

[58] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman, "Aggregation and degradation in JetStream: Streaming analytics in the wide area," in *NSDI*, 2014.

[59] B. Recht, C. Ré, S. J. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *NIPS*, 2011.

[60] P. Richtárik and M. Takác, "Distributed coordinate descent method for learning with big data," *CoRR*, 2013.

[61] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Cognitive modeling*, 1988.

[62] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet large scale visual recognition challenge," *IJCV*, 2015.

[63] O. Shamir, N. Srebro, and T. Zhang, "Communication-efficient distributed optimization using an approximate Newton-type method," in *ICML*, 2014.

[64] A. J. Smola and S. M. Narayanamurthy, "An architecture for parallel topic models," *VLDB*, 2010.

[65] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *CVPR*, 2015.

[66] M. Takác, A. S. Bijral, P. Richtárik, and N. Srebro, "Mini-batch primal and dual methods for SVMs," in *ICML*, 2013.

[67] TeleGeography, "Global Internet geography." https://www.telegeography.com/research-services/global-internet-geography/

[68] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. S. Sarma, R. Murthy, and H. Liu, "Data warehousing and analytics infrastructure at Facebook," in *SIGMOD*, 2010.

[69] K. I. Tsianos, S. F. Lawlor, and M. G. Rabbat, "Communication/computation tradeoffs in consensus-based distributed optimization," in *NIPS*, 2012.

[70] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, 1990.

[71] R. Viswanathan, A. Akella, and G. Ananthanarayanan, "Clarinet: WAN-aware optimization for analytics queries," in *OSDI*, 2016.

[72] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese, "WANalytics: Analytics for a geo-distributed data-intensive world," in *CIDR*, 2015.

[73] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese, "Global analytics in the face of bandwidth and regulatory constraints," in *NSDI*, 2015.

[74] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Managed communication and consistency for fast data-parallel iterative analytics," in *SoCC*, 2015.

[75] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang, "Moving objects databases: Issues and solutions," in *SSDBM*, 1998.

[76] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services," in *SOSP*, 2013.

[77] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: A new platform for distributed machine learning on big data," in *SIGKDD*, 2015.

[78] E. P. Xing, Q. Ho, P. Xie, and W. Dai, "Strategies and principles of distributed machine learning on big data," *CoRR*, 2015.

[79] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012.

[80] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "PrIter: A distributed framework for prioritized iterative computations," in *SoCC*, 2011.

[81] Y. Zhang, J. C. Duchi, and M. J. Wainwright, "Communication-efficient algorithms for statistical optimization," *JMLR*, 2013.

[82] Y. Zhang and X. Lin, "DiSCO: Distributed optimization for self-concordant empirical loss," in *ICML*, 2015.

[83] M. Zinkevich, A. J. Smola, and J. Langford, "Slow learners are fast," in *NIPS*, 2009.

[84] M. Zinkevich, M. Weimer, A. J. Smola, and L. Li, "Parallelized stochastic gradient descent," in *NIPS*, 2010.

Appendix

A. Convergence Proof of SGD under ASP

Stochastic Gradient Descent is a very popular algorithm, widely used for finding the minimizer/maximizer of a criterion (sum of differentiable functions) via iterative steps. The intuition behind the algorithm is that we randomly select an initial point $\boldsymbol{x}_0$ and keep moving toward the negative direction of the gradient, producing a sequence of points $\boldsymbol{x}_i, i = 1,...n$ until we detect that moving further decreases (increases) the minimization (maximization, respectively) criterion only negligibly.

Formally, step t of the SGD algorithm is defined as:

$$\boldsymbol{x}_t = \boldsymbol{x}_{t-1} - \eta_t \nabla f_t(\boldsymbol{x}_t) = \boldsymbol{x}_{t-1} - \eta_t \boldsymbol{g}_t = \boldsymbol{x}_{t-1} + \boldsymbol{u}_t \quad (1)$$

where η_t is the step size at step t, $\nabla f_t(\boldsymbol{x}_t)$ or $\boldsymbol{g}_t$ is the gradient at step t, and $\boldsymbol{u}_t = \eta_t \boldsymbol{g}_t$ is the update of step t.

Let us define an order of the updates up to step t. Suppose that the algorithm is distributed in D data centers, each of which uses P machines, and the logical clocks that mark progress start at 0. Then,

$$\boldsymbol{u}_t = \boldsymbol{u}_{d,p,c} = \boldsymbol{u}_{\lfloor \frac{t}{P} \rfloor \bmod D,\ t \bmod P,\ \lfloor \frac{t}{DP} \rfloor} \quad (2)$$

represents a mapping that loops through clocks ($c = \lfloor \frac{t}{DP} \rfloor$) and for each clock loops through data centers ($d = \lfloor \frac{t}{P} \rfloor \bmod D$) and for each data center loops through its workers ($p = t \bmod P$).

We now define a reference sequence of states that a single machine serial execution would go through if the updates were observed under the above ordering:

$$\boldsymbol{x}_t = \boldsymbol{x}_0 + \sum_{t'=1}^{t} \boldsymbol{u}_{t'} \quad (3)$$

Let Δ_c denote the threshold of mirror clock difference between different data centers. At clock c, let $A_{d,c}$ denote the $(c-\Delta_c)$-width window of updates at data center d: $A_{d,c} = [0, P-1] \times [0, c-\Delta_c-1]$. Also, let $K_{d,c}$ denote the subset of $A_{d,c}$ of significant updates (i.e., those broadcast to other data centers) and $L_{d,c}$ denote the subset of $A_{d,c}$ of the insignificant updates (not broadcast) from this data center. Clearly, $K_{d,c}$ and $L_{d,c}$ are disjoint and their union equals $A_{d,c}$.

Let s denote a user-chosen staleness threshold for SSP. Let $\tilde{\boldsymbol{x}}_t$ denote the sequence of noisy (i.e., inaccurate) *views* of the parameters $\boldsymbol{x}_t$. Let $B_{d,c}$ denote the $2s$-width window of updates at data center d: $B_{d,c} = [0, P-1] \times [c-s, c+s-1]$. A worker p in data center d will definitely see its own updates and may or may not see updates from other workers that belong to this window. Then, $M_{d,c}$ denotes the set of updates that are not seen in $\tilde{\boldsymbol{x}}_t$ and are seen in $\boldsymbol{x}_t$, whereas $N_{d,c}$ denotes the updates that are seen in $\tilde{\boldsymbol{x}}_t$ and not seen in $\boldsymbol{x}_t$. The sets $M_{d,c}$ and $N_{d,c}$ are disjoint and their union equals the set $B_{d,c}$.

We define the noisy view $\tilde{\boldsymbol{x}}_t$ using the above mapping:

$$\tilde{\boldsymbol{x}}_{d,p,c} = \sum_{p'=0}^{P-1} \sum_{c'=0}^{c-s-1} \boldsymbol{u}_{d,p',c'} + \sum_{c'=c-s}^{c-1} \boldsymbol{u}_{d,p,c'} + \sum_{(p',c') \in B'_{d,c} \subset B_{d,c}} \boldsymbol{u}_{d,p',c'} + \sum_{d' \neq d} \Big[\sum_{(p',c') \in K_{d',c'}} \boldsymbol{u}_{d',p',c'} \Big] \quad (4)$$

The difference between the reference view $\boldsymbol{x}_t$ and the noisy view $\tilde{\boldsymbol{x}}_t$ becomes:

$$\begin{aligned} \tilde{\boldsymbol{x}}_t - \boldsymbol{x}_t = \tilde{\boldsymbol{x}}_{d,p,c} - \boldsymbol{x}_t = \tilde{\boldsymbol{x}}_{\lfloor \frac{t}{P} \rfloor \bmod D, t \bmod P, \lfloor \frac{t}{DP} \rfloor} - \boldsymbol{x}_t = \\ - \sum_{i \in M_{d,c}} \boldsymbol{u}_i + \sum_{i \in N_{d,c}} \boldsymbol{u}_i - \sum_{d' \neq d} \sum_{i \in L_{d',c}} \boldsymbol{u}_i \\ + \sum_{d' \neq d} \Big[- \sum_{i \in M_{d',c}} \boldsymbol{u}_i + \sum_{i \in N_{d',c}} \boldsymbol{u}_i \Big] \end{aligned} \quad (5)$$

Finally, let $D(\boldsymbol{x}, \boldsymbol{x'})$ denote the distance between points $\boldsymbol{x}, \boldsymbol{x'} \in \mathbb{R}^n$:

$$D(\boldsymbol{x}, \boldsymbol{x'}) = \frac{1}{2} \|\boldsymbol{x} - \boldsymbol{x'}\|^2. \quad (6)$$

We now prove the following lemma:

Lemma. *For any* $\boldsymbol{x}^*, \tilde{\boldsymbol{x}}_t \in \mathbb{R}^n$,

$$\begin{aligned} \langle \tilde{\boldsymbol{x}}_t - \boldsymbol{x}^*, \tilde{\boldsymbol{g}}_t \rangle = \frac{1}{2} \eta_t \|\tilde{\boldsymbol{g}}_t\|^2 + \frac{D(\boldsymbol{x}^*, \boldsymbol{x}_t) - D(\boldsymbol{x}^*, \boldsymbol{x}_{t+1})}{\eta_t} \\ + \Big[- \sum_{i \in M_{d,c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle + \sum_{i \in N_{d,c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle \Big] \\ + \sum_{d' \neq d} \Big[- \sum_{i \in L_{d',c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle \Big] \\ + \sum_{d' \neq d} \Big[- \sum_{i \in M_{d',c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle + \sum_{i \in N_{d',c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle \Big] \end{aligned} \quad (7)$$

Proof.

$$\begin{aligned} D(\boldsymbol{x}^*, \boldsymbol{x}_{t+1}) - D(\boldsymbol{x}^*, \boldsymbol{x}_t) &= \frac{1}{2} \|\boldsymbol{x}^* - \boldsymbol{x}_{t+1}\|^2 - \frac{1}{2} \|\boldsymbol{x}^* - \boldsymbol{x}_t\|^2 \\ &= \frac{1}{2} \|\boldsymbol{x}^* - \boldsymbol{x}_t + \boldsymbol{x}_t - \boldsymbol{x}_{t+1}\|^2 - \frac{1}{2} \|\boldsymbol{x}^* - \boldsymbol{x}_t\|^2 \\ &= \frac{1}{2} \|\boldsymbol{x}^* - \boldsymbol{x}_t + \eta_t \tilde{\boldsymbol{g}}_t\|^2 - \frac{1}{2} \|\boldsymbol{x}^* - \boldsymbol{x}_t\|^2 \\ &= \frac{1}{2} \langle \boldsymbol{x}^* - \boldsymbol{x}_t + \eta_t \tilde{\boldsymbol{g}}_t, \boldsymbol{x}^* - \boldsymbol{x}_t + \eta_t \tilde{\boldsymbol{g}}_t \rangle - \frac{1}{2} \langle \boldsymbol{x}^* - \boldsymbol{x}_t, \boldsymbol{x}^* - \boldsymbol{x}_t \rangle \\ &= \frac{1}{2} \langle \boldsymbol{x}^* - \boldsymbol{x}_t, \boldsymbol{x}^* - \boldsymbol{x}_t \rangle + \frac{1}{2} \langle \eta_t \tilde{\boldsymbol{g}}_t, \eta_t \tilde{\boldsymbol{g}}_t \rangle + \langle \boldsymbol{x}^* - \boldsymbol{x}_t, \eta_t \tilde{\boldsymbol{g}}_t \rangle \\ &\quad - \frac{1}{2} \langle \boldsymbol{x}^* - \boldsymbol{x}_t, \boldsymbol{x}^* - \boldsymbol{x}_t \rangle \\ &= \frac{1}{2} {\eta_t}^2 \|\tilde{\boldsymbol{g}}_t\|^2 + \eta_t \langle \boldsymbol{x}^* - \boldsymbol{x}_t, \tilde{\boldsymbol{g}}_t \rangle \\ &= \frac{1}{2} {\eta_t}^2 \|\tilde{\boldsymbol{g}}_t\|^2 - \eta_t \langle \boldsymbol{x}_t - \boldsymbol{x}^*, \tilde{\boldsymbol{g}}_t \rangle \\ &= \frac{1}{2} {\eta_t}^2 \|\tilde{\boldsymbol{g}}_t\|^2 - \eta_t \langle \boldsymbol{x}_t - \tilde{\boldsymbol{x}}_t + \tilde{\boldsymbol{x}}_t - \boldsymbol{x}^* \rangle \\ &= \frac{1}{2} {\eta_t}^2 \|\tilde{\boldsymbol{g}}_t\|^2 - \eta_t \langle \boldsymbol{x}_t - \tilde{\boldsymbol{x}}_t, \tilde{\boldsymbol{g}}_t \rangle - \eta_t \langle \tilde{\boldsymbol{x}}_t - \boldsymbol{x}^*, \tilde{\boldsymbol{g}}_t \rangle \implies \end{aligned}$$

$$\langle \tilde{\boldsymbol{x}}_t - \boldsymbol{x}^*, \tilde{\boldsymbol{g}}_t \rangle = \frac{1}{2}\eta_t \|\tilde{\boldsymbol{g}}_t\|^2 + \frac{D(\boldsymbol{x}^*, \boldsymbol{x}_t) - D(\boldsymbol{x}^*, \boldsymbol{x}_{t+1})}{\eta_t} - \langle \boldsymbol{x}_t - \tilde{\boldsymbol{x}}_t, \tilde{\boldsymbol{g}}_t \rangle \quad (8)$$

Substituting the RHS of Equation 5 into Equation 8 completes the proof. □

Theorem 1. ***(Convergence of SGD under ASP).***

Suppose that, in order to compute the minimizer x^* of a convex function $f(\boldsymbol{x}) = \sum_{t=1}^{T} f_t(\boldsymbol{x})$, with $f_t, t = 1, 2, \ldots, T$, convex, we use stochastic gradient descent on one component ∇f_t at a time. Suppose also that 1) the algorithm is distributed in D data centers, each of which uses P machines, 2) within each data center, the SSP protocol is used, with a fixed staleness of s, and 3) a fixed mirror clock difference Δ_c is allowed between any two data centers. Let $\boldsymbol{u}_t = -\eta_t \nabla f_t(\tilde{\boldsymbol{x}}_t)$, where the step size η_t decreases as $\eta_t = \frac{\eta}{\sqrt{t}}$ and the significance threshold v_t decreases as $v_t = \frac{v}{\sqrt{t}}$. If we further assume that: $\|\nabla f_t(\boldsymbol{x})\| \leq L$, $\forall \boldsymbol{x} \in dom(f_t)$ and $\max(D(\boldsymbol{x}, \boldsymbol{x}')) \leq \Delta^2, \forall \boldsymbol{x}, \boldsymbol{x}' \in dom(f_t)$. Then, as $T \to \infty$,

$$R[X] = \sum_{t=1}^{T} f_t(\tilde{\boldsymbol{x}}_t) - f(\boldsymbol{x}^*) = O(\sqrt{T})$$

and therefore

$$\lim_{T \to \infty} \frac{R[X]}{T} \to 0$$

Proof.

$$\begin{aligned}
R[X] &= \sum_{t=1}^{T} f_t(\tilde{\boldsymbol{x}}_t) - f_t(\boldsymbol{x}^*) \\
&\leq \sum_{t=1}^{T} \langle \nabla f_t(\tilde{\boldsymbol{x}}_t), \tilde{\boldsymbol{x}}_t - \boldsymbol{x}^* \rangle \quad \text{(convexity of } f_t\text{)} \\
&= \sum_{t=1}^{T} \langle \tilde{\boldsymbol{g}}_t, \tilde{\boldsymbol{x}}_t - \boldsymbol{x}^* \rangle \\
&= \sum_{t=1}^{T} \Bigg[\frac{1}{2}\eta_t \|\tilde{\boldsymbol{g}}_t\|^2 + \frac{D(\boldsymbol{x}^*, \boldsymbol{x}_t) - D(\boldsymbol{x}^*, \boldsymbol{x}_{t+1})}{\eta_t} \\
&\quad + \sum_{d' \neq d} \Big[- \sum_{i \in L_{d',c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle \Big] \\
&\quad + \Big[- \sum_{i \in M_{d,c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle + \sum_{i \in N_{d,c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle \Big] \\
&\quad + \sum_{d' \neq d} \Big[- \sum_{i \in M_{d',c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle + \sum_{i \in N_{d',c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle \Big] \Bigg] \quad (9)
\end{aligned}$$

We first bound the upper limit of the term: $\sum_{t=1}^{T} \frac{1}{2}\eta_t \|\tilde{\boldsymbol{g}}_t\|^2$:

$$\begin{aligned}
\sum_{t=1}^{T} \frac{1}{2}\eta_t \|\tilde{\boldsymbol{g}}_t\|^2 &\leq \sum_{t=1}^{T} \frac{1}{2}\eta_t L^2 \qquad (\|\nabla f_t(\boldsymbol{x})\| \leq L) \\
&= \sum_{t=1}^{T} \frac{1}{2}\frac{\eta}{\sqrt{t}} L^2 \\
&= \frac{1}{2}\eta L^2 \sum_{t=1}^{T} \frac{1}{\sqrt{t}} \qquad (\sum_{t=1}^{T} \frac{1}{\sqrt{t}} \leq 2\sqrt{T}) \\
&\leq \frac{1}{2}\eta L^2 2\sqrt{T} = \eta L^2 \sqrt{T} \quad (10)
\end{aligned}$$

Second, we bound the upper limit of the term: $\sum_{t=1}^{T} \frac{D(x^*, x_t) - D(x^*, x_{t+1})}{\eta_t}$:

$$\begin{aligned}
&\sum_{t=1}^{T} \frac{D(\boldsymbol{x}^*, \boldsymbol{x}_t) - D(\boldsymbol{x}^*, \boldsymbol{x}_{t+1})}{\eta_t} \\
&= \frac{D(\boldsymbol{x}^*, \boldsymbol{x}_1)}{\eta_1} - \frac{D(\boldsymbol{x}^*, \boldsymbol{x}_{T+1})}{\eta_T} + \sum_{t=2}^{T} D(\boldsymbol{x}^*, \boldsymbol{x}_t)(\frac{1}{\eta_t} - \frac{1}{\eta_{t-1}}) \\
&\leq \frac{\Delta^2}{\eta} - 0 + \frac{\Delta^2}{\eta} \sum_{t=2}^{T} [\sqrt{t} - \sqrt{t-1}] \quad (\max(D(\boldsymbol{x}, \boldsymbol{x}')) \leq \Delta^2) \\
&= \frac{\Delta^2}{\eta} + \frac{\Delta^2}{\eta}[\sqrt{T} - 1] \\
&= \frac{\Delta^2}{\eta}\sqrt{T} \quad (11)
\end{aligned}$$

Third, we bound the upper limit of the term: $\sum_{t=1}^{T} \sum_{d' \neq d} \Big[- \sum_{i \in L_{d',c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle \Big]$:

$$\begin{aligned}
&\sum_{t=1}^{T} \sum_{d' \neq d} \Big[- \sum_{i \in L_{d',c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle \Big] \\
&\leq \sum_{t=1}^{T} (D-1) \Big[- \sum_{i \in L_{d',c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle \Big] \leq \sum_{t=1}^{T} (D-1) v_t \\
&= (D-1) \sum_{t=1}^{T} \frac{v}{\sqrt{t - (s + \Delta_c + 1)P}} \\
&\leq (D-1) v \sum_{t=(s+\Delta_c+1)P+1}^{T} \frac{1}{\sqrt{T - (s + \Delta_c + 1)P}} \\
&\leq 2(D-1) v \sqrt{T - (s + \Delta_c + 1)P} \\
&\leq 2(D-1) v \sqrt{T} \\
&\leq 2Dv\sqrt{T} \quad (12)
\end{aligned}$$

where the fourth inequality follows from the fact that: $\sum_{t=(s+\Delta_c+1)P+1}^{T} \frac{1}{\sqrt{T-(s+\Delta_c+1)P}} \leq \sqrt{T - (s + \Delta_c + 1)P}$.

Fourth, we bound the upper limit of the term: $\sum_{t=1}^{T} \Big[- \sum_{i \in M_{d,c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle + \sum_{i \in N_{d,c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle \Big]$:

$$
\begin{aligned}
&\sum_{t=1}^{T}\Big[-\sum_{i\in M_{d,c}}\eta_i\langle\tilde{\boldsymbol{g}}_i,\tilde{\boldsymbol{g}}_t\rangle+\sum_{i\in N_{d,c}}\eta_i\langle\tilde{\boldsymbol{g}}_i,\tilde{\boldsymbol{g}}_t\rangle\Big]\\
&\le\sum_{t=1}^{T}[|M_{d,c}|+|N_{d,c}\,|]\eta_{max(1,t-(s+1)P)}L^2\\
&=L^2\Big[\sum_{t=1}^{(s+1)P}[|M_{d,c}|+|N_{d,c}|]\eta\\
&\quad+\sum_{t=(s+1)P+1}^{T}[|M_{d,c}|+|N_{d,c}|]\eta_{t-(s+1)P}\Big]\\
&=L^2\Big[\sum_{t=1}^{(s+1)P}[|M_{d,c}|+|N_{d,c}|]\eta\\
&\quad+\sum_{t=(s+1)P+1}^{T}[|M_{d,c}|+|N_{d,c}|]\frac{\eta}{\sqrt{t-(s+1)P}}\Big]\\
&\le\eta L^2\Big[\sum_{t=1}^{(s+1)P}2s(P-1)\\
&\quad+\sum_{t=(s+1)P+1}^{T}2s(P-1)\frac{1}{\sqrt{t-(s+1)P}}\Big]\\
&=2\eta L^2s(P-1)\Big[(s+1)P+\sum_{t=(s+1)P+1}^{T}\frac{1}{\sqrt{T-(s+1)P}}\Big]\\
&\le2\eta L^2s(P-1)\Big[(s+1)P+2\sqrt{T-(s+1)P}\Big]\\
&\le2\eta L^2s(P-1)[(s+1)P+2\sqrt{T}]\\
&=2\eta L^2s(s+1)(P-1)P+4\eta L^2s(P-1)\sqrt{T}\\
&\le2\eta L^2(s+1)(s+1)(P-1)P+4\eta L^2(s+1)(P-1)\sqrt{T}\\
&=2\eta L^2(s+1)^2(P-1)P+4\eta L^2(s+1)(P-1)\sqrt{T}\\
&\le2\eta L^2(s+1)^2PP+4\eta L^2(s+1)P\sqrt{T}\\
&=2\eta L^2[(s+1)P]^2+4\eta L^2(s+1)P\sqrt{T}\qquad(13)
\end{aligned}
$$

where the first inequality follows from the fact that $\eta_{max(1,t-(s+1)P)}\ge\eta_t, t\in M_{d,t}\cup N_{d,t}$, the second inequality follows from the fact that $|M_{d,t}|+|N_{d,t}|\le 2s(P-1)$, and the third inequality follows from the fact that $\sum_{t=(s+1)P+1}^{T}\Big[\frac{1}{\sqrt{T-(s+1)P}}\Big]\le 2\sqrt{T-(s+1)P}$.

Similarly, $\forall d'\in D'=D\setminus\{d\}$, we can prove that:

$$
\begin{aligned}
&\sum_{t=1}^{T}\Big[-\sum_{i\in M_{d',c}}\eta_i\langle\tilde{\boldsymbol{g}}_i,\tilde{\boldsymbol{g}}_t\rangle+\sum_{i\in N_{d',c}}\eta_i\langle\tilde{\boldsymbol{g}}_i,\tilde{\boldsymbol{g}}_t\rangle\Big]\le\\
&\quad2\eta L^2[(s+\Delta_c+1)P]^2+4\eta L^2(s+\Delta_c+1)P\sqrt{T}
\end{aligned}
$$

which implies:

$$
\begin{aligned}
&\sum_{t=1}^{T}\sum_{d'\neq d}\Big[-\sum_{i\in M_{d',c}}\boldsymbol{u}_i+\sum_{i\in N_{d',c}}\boldsymbol{u}_i\Big]\le\\
&\quad D\Big[2\eta L^2[(s+\Delta_c+1)P]^2+4\eta L^2(s+\Delta_c+1)P\sqrt{T}\Big]
\end{aligned}
$$

By combining all the above upper bounds, we have:

$$
\begin{aligned}
R[X]&\le\eta L^2\sqrt{T}+\frac{\Delta^2}{\eta}\sqrt{T}+2Dv\sqrt{T}+2\eta L^2[(s+1)P]^2\\
&\quad+4\eta L^2(s+1)P\sqrt{T}\\
&\quad+D\Big[2\eta L^2[(s+\Delta_c+1)P]^2+4\eta L^2(s+\Delta_c+1)P\sqrt{T}\Big]\\
&=O(\sqrt{T})\qquad(14)
\end{aligned}
$$

and thus $\lim_{T\to\infty}\frac{R[X]}{T}\to 0$. □

B. Performance Results of SSP

Due to limited space, we present the performance results of SSP for *MF (Matrix Factorization)* and *TM (Topic Modeling)* here. We do not present the results of SSP for *IC (Image Classification)* because SSP has worse performance than BSP for *IC* [18]. In our evaluation, BSP and SSP are used among all worker machines for `Baseline` and `LAN`, whereas for `Gaia`, they are used only within each data center. To show the performance difference between BSP and SSP, we show both results together.

B.1. SSP Performance on EC2 Deployment

Similar to Section 6.1, Figures 13 and 14 show the execution time until convergence for *MF* and *TM*, normalized to `Baseline` with BSP on EC2. The data label above each bar shows the speedup over `Baseline` for *the respective*

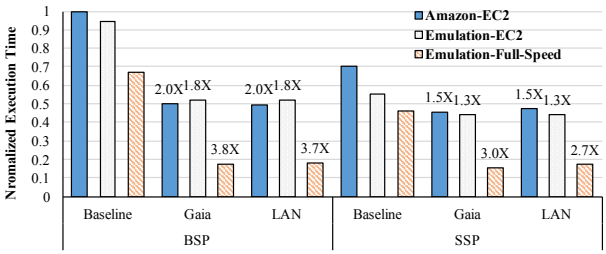


Figure 13: Normalized execution time of *MF* until convergence when deployed across 11 EC2 regions

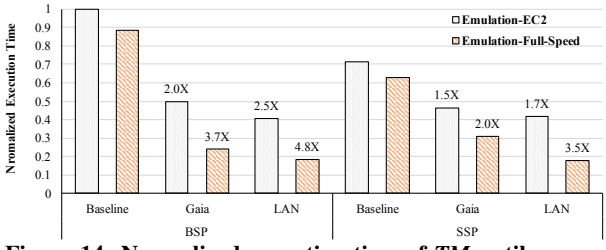


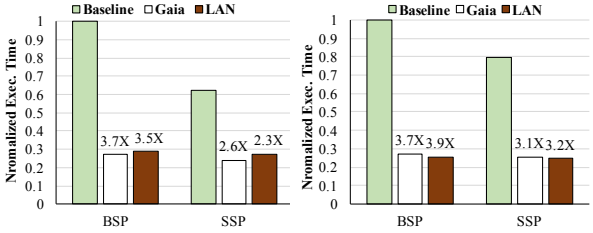
Figure 14: Normalized execution time of *TM* until convergence when deployed across 11 EC2 regions

We see that `Gaia` significantly improves the performance of `Baseline` with SSP. For *MF*, `Gaia` provides a speedup of 1.3–3.0× over `Baseline` with SSP, and successfully approaches the speedups of `LAN` with SSP. For *TM*, `Gaia` achieves speedups of 1.5–2.0× over `Baseline`. Note that for *TM*, `Gaia` with BSP outperforms `Gaia` with SSP. The reason is that SSP allows using stale, and thus

inaccurate, values in order to get the benefit of more efficient communication. However, compared to `Baseline`, the benefit of employing SSP to reduce communication overhead is much smaller for `Gaia` because it uses SSP only to synchronize a small number of machines within a data center. Thus, the cost of inaccuracy outweighs the benefit of SSP in this case. Fortunately, `Gaia` decouples the synchronization model within a data center from the synchronization model across different data centers. Thus, we can freely choose the combination of synchronization models that works better for `Gaia`.

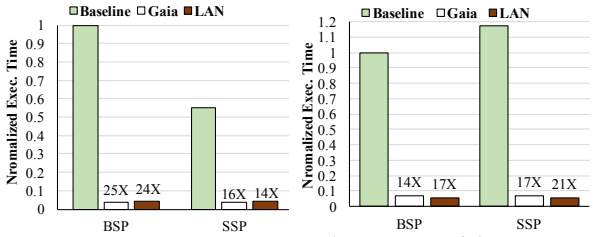
B.2. SSP Performance and WAN Bandwidth

Similar to Section 6.2, Figures 15 and 16 show the normalized execution time until convergence on two deployments: V/C WAN and S/S WAN. The data label above each bar shows the speedup over `Baseline` for *the respective deployment and synchronization model*.



(a) *Matrix Factorization (MF)* (b) *Topic Modeling (TM)*

Figure 15: Normalized execution time until convergence with the WAN bandwidth between Virginia and California



(a) *Matrix Factorization (MF)* (b) *Topic Modeling (TM)*

Figure 16: Normalized execution time until convergence with the WAN bandwidth between Singapore and São Paulo

We find that `Gaia` performs very well compared to `Baseline` with SSP in both high WAN bandwidth (V/C WAN) and low WAN bandwidth (S/S WAN) settings. For V/C WAN, `Gaia` achieves a speedup of 2.6× for *MF* and 3.1× for *TM* over `Baseline` with SSP. For both applications, the performance of `Gaia` is almost the same as the performance of `LAN`. For S/S WAN, `Gaia` provides a speedup of 15.7× / 16.8× for *MF* / *TM* over `Baseline` with SSP, and successfully approaches the `LAN` speedups. We conclude that `Gaia` provides significant performance improvement over `Baseline`, irrespective of the synchronization model used by `Baseline`.

C. EC2 Cost Model Details

We use the on-demand pricing of Amazon EC2 published for January 2017 as our cost model [8]. As the pricing might change over time, we provide the details of the cost model in Table 2.

The CPU instance is `c4.4xlarge` or `m4.4xlarge`, depending on the availability in each EC2 region. The GPU instance is `g2.8xlarge`. The low-cost instance (`m4.xlarge`) is the one used for centralizing input data. All the instance costs are shown in USD per hour. All WAN data transfer costs are shown in USD per GB.

Table 2: Cost model details

Region	CPU Instance	GPU Instance	Low-cost Instance	Send to WANs	Recv. from WANs
Virginia	$0.86	$2.60	$0.22	$0.02	$0.01
California	$1.01	$2.81	$0.22	$0.02	$0.01
Oregon	$0.86	$2.60	$0.22	$0.02	$0.01
Ireland	$0.95	$2.81	$0.24	$0.02	$0.01
Frankfurt	$1.03	$3.09	$0.26	$0.02	$0.01
Tokyo	$1.11	$3.59	$0.27	$0.09	$0.01
Seoul	$1.06	$3.59	$0.28	$0.08	$0.01
Singapore	$1.07	$4.00	$0.27	$0.09	$0.01
Sydney	$1.08	$3.59	$0.27	$0.14	$0.01
Mumbai	$1.05	$3.59	$0.26	$0.09	$0.01
São Paulo	$1.37	$4.00	$0.34	$0.16	$0.01

Efficient Memory Disaggregation with INFINISWAP

Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, Kang G. Shin
University of Michigan

Abstract

Memory-intensive applications suffer large performance loss when their working sets do not fully fit in memory. Yet, they cannot leverage otherwise unused remote memory when paging out to disks even in the presence of large imbalance in memory utilizations across a cluster. Existing proposals for memory disaggregation call for new architectures, new hardware designs, and/or new programming models, making them infeasible.

This paper describes the design and implementation of INFINISWAP, a remote memory paging system designed specifically for an RDMA network. INFINISWAP opportunistically harvests and transparently exposes unused memory to unmodified applications by dividing the swap space of each machine into many slabs and distributing them across many machines' remote memory. Because one-sided RDMA operations bypass remote CPUs, INFINISWAP leverages the power of many choices to perform decentralized slab placements and evictions.

We have implemented and deployed INFINISWAP on an RDMA cluster without any modifications to user applications or the OS and evaluated its effectiveness using multiple workloads running on unmodified VoltDB, Memcached, PowerGraph, GraphX, and Apache Spark. Using INFINISWAP, throughputs of these applications improve between $4\times$ ($0.94\times$) to $15.4\times$ ($7.8\times$) over disk (Mellanox nbdX), and median and tail latencies between $5.4\times$ ($2\times$) and $61\times$ ($2.3\times$). INFINISWAP achieves these with negligible remote CPU usage, whereas nbdX becomes CPU-bound. INFINISWAP increases the overall memory utilization of a cluster and works well at scale.

1 Introduction

Memory-intensive applications [18, 20, 23, 77] are widely used today for low-latency services and data-intensive analytics alike. The main reason for their popularity is simple: as long as requests are served from memory and disk accesses are minimized, latency decreases and throughput increases. However, these applications experience rapid performance deteriorations when their working sets do not fully fit in memory (§2.2).

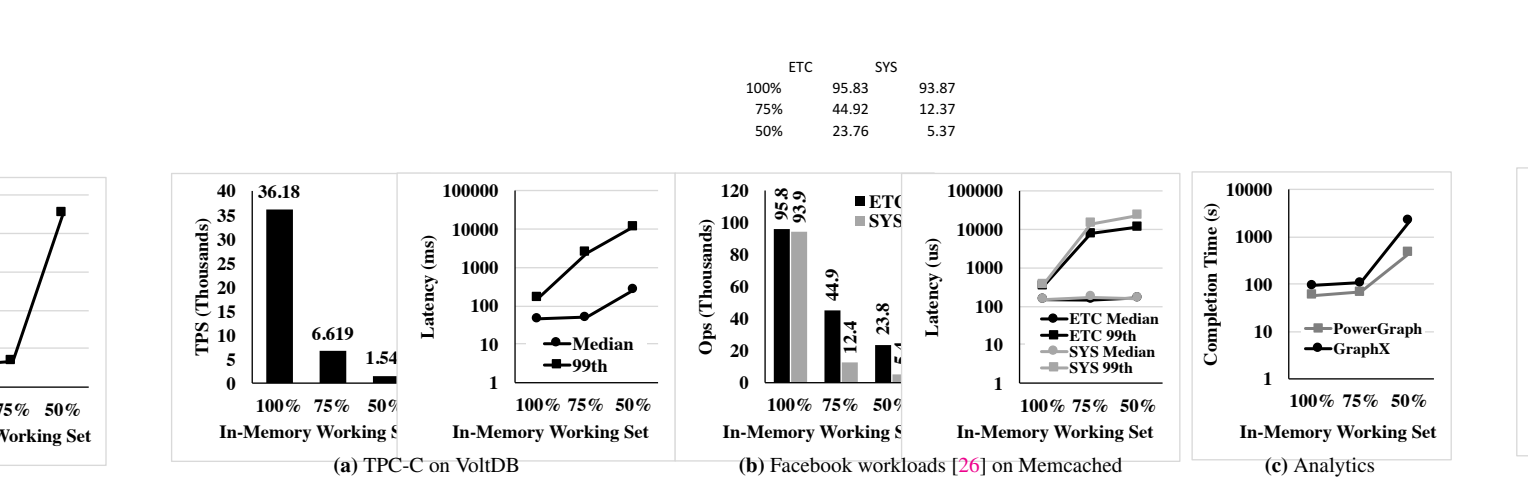
There are two primary ways of mitigating this issue: (i) rightsizing memory allocation and (ii) increasing the effective memory capacity of each machine. Rightsizing is difficult because applications often overestimate their requirements [71] or attempt to allocate for peak usage [28], resulting in severe underutilization and unbalanced memory usage across the cluster. Our analysis of two large production clusters shows that more than 70% of the time there exists severe imbalance in memory utilizations across their machines (§2.3).

Proposals for memory disaggregation [42, 49, 53, 70] acknowledge this imbalance and aim to expose a global memory bank to all machines to increase their effective memory capacities. Recent studies suggest that modern RDMA networks can meet the latency requirements of memory disaggregation architectures for numerous in-memory workloads [42, 70]. However, existing proposals for memory disaggregation call for new architectures [11, 12, 49], new hardware designs [56, 57], and new programming models [63, 69], rendering them infeasible.

In this paper, we present INFINISWAP, a new scalable, decentralized remote memory paging solution that enables efficient memory disaggregation. It is designed specifically for RDMA networks to perform remote memory paging when applications cannot fit their working sets in local memory. It does so *without* requiring any coordination or modifications to the underlying infrastructure, operating systems, and applications (§3).

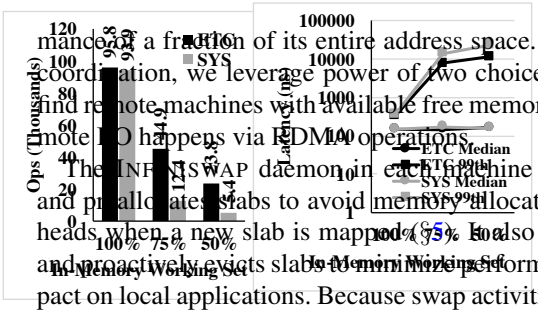
INFINISWAP is not the first to exploit memory imbalance and disk-network latency gap for remote memory paging [2, 25, 31, 37, 40, 41, 55, 58, 64]. However, unlike existing solutions, it does not incur high remote CPU overheads, scalability concerns from central coordination to find machines with free memory, and large performance loss due to evictions from, and failures of, remote memory.

INFINISWAP addresses these challenges via two primary components: a block device that is used as the swap space and a daemon that manages remotely accessible memory. Both are present in every machine and work together without any central coordination. The INFINISWAP block device writes synchronously to remote memory for low latency and asynchronously to disk for fault-tolerance (§4). To mitigate high recovery overheads of disks in the presence of remote evictions and failures, we divide its address space into fixed-size slabs and place them across many machines' remote memory. As a result, remote evictions and failures only affect the perfor-



(a) TPC-C on VoltDB (b) Facebook workloads [26] on Memcached (c) Analytics

Figure 1: For modern in-memory applications, a decrease in the percentage of the working set that fits in memory often results in a disproportionately larger loss of performance. This effect is further amplified for tail latencies. All plots show single-machine performance and the median value of five runs. Lower is better for the latency-related plots (lines) and the opposite holds for the throughput-related ones (bars). Note the logarithmic Y-axes in the latency/completion time plots.



manage a fraction of its entire address space. To avoid coordination, we leverage power of two choices [62] to find remote machines with available free memory. All remote IO happens via RDMA operations.

The INFINISWAP daemon in each machine monitors and pre-allocates slabs to avoid memory allocation overheads when a new slab is mapped (§5). It also monitors and proactively evicts slabs to minimize performance impact on local applications. Because swap activities on the hosted slabs are transparent to the daemon, we leverage power of many choices [68] to perform batch eviction without any central coordination.

We have implemented INFINISWAP on Linux kernel 3.13.0 (§6) and deployed it on a 56 Gbps, 32-machine RDMA cluster on CloudLab [5]. We evaluated it using multiple unmodified memory-intensive applications: VoltDB, Memcached, PowerGraph, GraphX, and Apache Spark using industrial benchmarks and production workloads (§7). Using INFINISWAP, throughputs improve between 4× (0.94×) and 15.4× (7.8×) over disk (Mellanox nbdX [2]), and median and tail latencies by up to 5.4× (2×) and 61× (2.3×), respectively. Memory-heavy workloads experience limited performance difference during paging, while CPU-heavy workloads experience some degradation. In comparison to nbdX, INFINISWAP does not use any remote CPU and provides a 2×–4× higher read/write bandwidth. INFINISWAP can recover from remote evictions and failures while still providing higher application-level performance in comparison to disks. Finally, its benefits hold in the presence of high concurrency and at scale, with a negligible increase in network bandwidth usage.

Despite its effectiveness, INFINISWAP cannot transparently emulate memory disaggregation for CPU-heavy workloads such as Spark and VoltDB (unlike memory-intensive Memcached and PowerGraph) due to the inherent overheads of paging (e.g., context switching). We still consider it useful because of its other tangible benefits. For example, when working sets do not fit in memory, VoltDB's performance degrades linearly using INFINISWAP instead of experiencing a super-linear drop.

We discuss related work in Section 9.

2 Motivation

This section overviews necessary background (§2.1) and discusses potential benefits from paging to remote memory in memory-intensive workloads (§2.2) as well as opportunities for doing so in production clusters (§2.3).

2.1 Background

Paging. Modern operating systems (OSes) support virtual memory to provide applications with larger address spaces than physically possible, using fixed-size pages (typically 4KB) as the unit of memory management. Usually, there are many more virtual pages than physical ones. Page faults occur whenever an application addresses a virtual address, whose corresponding page does not reside in physical memory. Subsequently, the virtual memory manager (VMM) consults with the page table to bring that page into memory; this is known as *paging in*. To make space for the new page, the VMM may need to *page out* one or more already existing pages to a block device, which is known as the *swap space*.

Any block device that implements an expected interface can be used as a swap space. INFINISWAP is written as a virtual block device to perform this role.

See [14] for a detailed description of memory management and its many optimizations in a modern OS.

Application Deployment Model. We consider a container-based application deployment model, which is common in production datacenters [28, 71, 76] as well as in container-as-a-service (CaaS) models [4, 8, 13, 50]. These clusters use resource allocation or scheduling algorithms [28, 43, 47, 73] to determine resource shares of different applications and deploy application processes

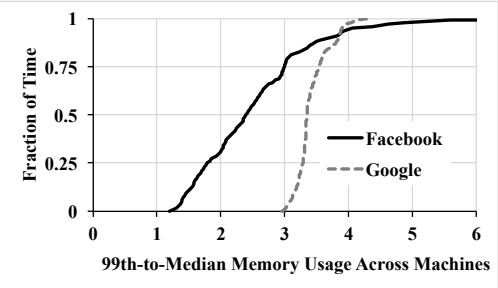


Figure 2: Imbalance in 10s-averaged memory usage in two large production clusters at Facebook and Google.

in containers to ensure resource isolation. Applications start paging when they require more memory than the memory limits of their containers.

Network Model. INFINISWAP requires a low-latency, RDMA network, but we do not make any assumptions about specific RDMA technologies (e.g., Infiniband vs. RoCE) or network diameters. Although we evaluate INFINISWAP in a small-scale environment, recent results suggest that deploying RDMA (thus INFINISWAP) on large datacenters may indeed be feasible [48, 61, 79].

2.2 Potential Benefits

To illustrate the adverse effects of paging, we consider four application types: (i) a standard TPC-C benchmark [22] running on the VoltDB [23] in-memory database; (ii) two Facebook-like workloads [26] running on the Memcached [18] key-value store; (iii) PowerGraph [45] running the TunkRank algorithm [1] on a Twitter dataset [52]; and (iv) PageRank running on Apache Spark [77] and GraphX [46] on the Twitter dataset. We found that Spark starts thrashing during paging and does not complete in many cases. We defer discussion of Spark to Section 7.2.4 and consider the rest here.

To avoid externalities, we only focus on single-server performance. Peak memory usage of each of these runs were around 10GB, significantly smaller than the server's total physical memory. We run each application inside containers of different memory capacities: $x\%$ in the X-axes of Figure 1 refers to a run inside a container that can hold at most $x\%$ of the application's working set in memory, and $x < 100$ forces paging. Section 7.2 has more details on the experimental setups.

We highlight two observations that show large potential benefits from INFINISWAP. First, paging has significant, non-linear impact on performance (Figure 1). For example, a 25% reduction in in-memory working set results in a $5.5\times$ and $2.1\times$ throughput loss for VoltDB and Memcached; in contrast, PowerGraph and GraphX worsen marginally. However, another 25% reduction makes VoltDB, Memcached, PowerGraph, and GraphX up to $24\times$, $17\times$, $8\times$, and $23\times$ worse, respectively.

Second, paging implications are highlighed particularly at the tail latencies. As working sets do not fit into memory, the 99th-percentile latencies of VoltDB and Memcached worsen by up to $71.5\times$ and $21.5\times$, respectively. In contrast, their median latencies worsens by up to $5.7\times$ and $1.1\times$, respectively.

These gigantic performance gaps suggest that a theoretical, 100%-efficient memory disaggregation solution can result in huge benefits, assuming that everything such solutions may require is ensured. It also shows that bridging some of these gaps by a practical, deployable solution can be worthwhile.

2.3 Characteristics of Memory Imbalance

To understand the presence of memory imbalance in modern clusters and corresponding opportunities, we analyzed traces from two production clusters: (i) a 3000-machine data analytics cluster (Facebook) and (ii) a 12500-machine cluster (Google) running a mix of diverse short- and long-running applications.

We highlight two key observations – the presence of memory imbalance and its temporal variabilities – that guide INFINISWAP's design decisions.

Presence of Imbalance. We found that the memory usage across machines can be substantially unbalanced in the short term (e.g., tens of seconds). Causes of imbalance include placement and scheduling constraints [28, 44] and resource fragmentation during packing [47, 76], among others. We measured memory utilization imbalance by calculating the 99th-percentile to the median usage ratio over 10-second intervals (Figure 2). With a perfect balance, these values would be 1. However, we found this ratio to be 2.4 in Facebook and 3.35 in Google more than half the time; meaning, most of the time, more than a half of the cluster aggregate memory remains unutilized.

Temporal Variabilities. Although skewed, memory utilizations remained stable over short intervals, which is useful for predictable decision-making when selecting remote machines. To analyze the stability of memory utilizations, we adopted the methodology described by Chowdhury et al. [32, §4.3]. Specifically, we consider a machine's memory utilization $U_t(m)$ at time t to be stable for the duration T if the difference between $U_t(m)$ and the average value of $U_t(m)$ over the interval $[t, t+T)$ remains within 10% of $U_t(m)$. We observed that average memory utilizations of a machine remained stable for smaller durations with very high probabilities. For the most unpredictable machine in the Facebook cluster, the probabilities that its current memory utilization from any instant will not change by more than 10% for the next 10, 20, and 40 seconds were 0.74, 0.58, and 0.42, respectively. For Google, the corresponding numbers were 0.97, 0.94, and 0.89, respectively. We believe

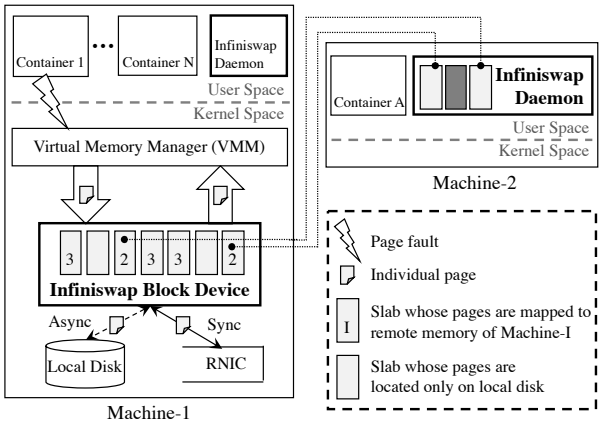


Figure 3: INFINISWAP architecture. Each machine loads a block device as a kernel module (set as swap device) and runs an INFINISWAP daemon. The block device divides its address space into slabs and transparently maps them across many machines' remote memory; paging happens at page granularity via RDMA.

that the higher probabilities in the Google cluster are due to its long-running services, whereas the Facebook cluster runs data analytics with many short tasks [24].

3 INFINISWAP Overview

INFINISWAP is a decentralized memory disaggregation solution for RDMA clusters that opportunistically uses remote memory for paging. In this section, we present a high-level overview of INFINISWAP to help the reader follow how INFINISWAP performs efficient and fault-tolerant memory disaggregation (§4), how it enables fast and transparent memory reclamation (§5), and its implementation details (§6).

3.1 Problem Statement

The main goal of INFINISWAP is to *efficiently expose all of a cluster's memory to user applications* without any modifications to those applications or the OSes of individual machines. It must also be *scalable, fault-tolerant, and transparent* so that application performance on remote machines remains unaffected.

3.2 Architectural Overview

INFINISWAP consists of two primary components – INFINISWAP block device and INFINISWAP daemon – that are present in every machine and work together without any central coordination (Figure 3).

The INFINISWAP block device exposes a conventional block device I/O interface to the virtual memory manager (VMM), which treats it as a fixed-size swap partition. The entire address space of this device is logically partitioned into fixed-size *slabs* (SlabSize). A slab is the unit of remote mapping and load balancing in INFINISWAP. Slabs from the same device can be mapped to multiple remote machines' memory for performance and load balancing (§4.2). All pages belonging to the same slab are mapped to the same remote machine. On the INFINISWAP daemon side, a slab is a physical memory chunk of SlabSize that is mapped to and used by an INFINISWAP block device as remote memory.

If a slab is mapped to remote memory, INFINISWAP synchronously writes a page-out request for that slab to remote memory using RDMA WRITE, while writing it asynchronously to the local disk. If it is not mapped, INFINISWAP synchronously writes the page only to the local disk. For page-in requests or reads, INFINISWAP consults the slab mapping to read from the appropriate source; it uses RDMA READ for remote memory.

The INFINISWAP daemon runs in the user space and only participates in control plane activities. Specifically, it responds to slab-mapping requests from INFINISWAP block devices, preallocates its local memory when possible to minimize time overheads in slab-mapping initialization, and proactively evicts slabs, when necessary, to ensure minimal impact on local applications. All control plane communications take place using RDMA SEND/RECV.

We have implemented INFINISWAP as a loadable kernel module for Linux 3.13.0 and deployed it in a 32-machine RDMA cluster. It performs well for a large variety of memory-intensive workloads (§7.2).

Scalability. INFINISWAP leverages the well-known power-of-choices techniques [62, 68] during both slab placement in block devices (§4.2) and eviction in daemons (§5.2). The reliance on decentralized techniques makes INFINISWAP more scalable by avoiding the need for constant coordination, while still achieving low-latency mapping and eviction.

Fault-tolerance. Because INFINISWAP does not have a central coordinator, it does not have a single point of failure. If a remote machine fails or becomes unreachable, INFINISWAP relies on the remaining remote memory and the local backup disk (§4.5). If the local disk also fails, INFINISWAP provides the same failure semantic as of today.

4 Efficient Memory Disaggregation via INFINISWAP Block Device

In this section, we describe how INFINISWAP block devices manage their address spaces (§4.1), perform decentralized slab placement to ensure better performance and load balancing (§4.2), handle I/O requests (§4.3), and minimize the impacts of slab evictions (§4.4) and remote failures (§4.5).

4.1 Slab Management

An INFINISWAP block device logically divides its entire address space into multiple slabs of fixed size (SlabSize).

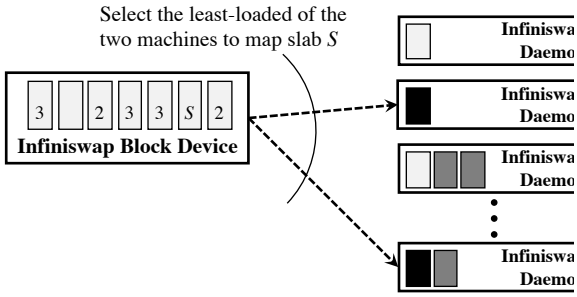


Figure 4: INFINISWAP block device uses power of two choi to select machines with the most available memory. It pref machines without any of its slabs to those who have to c tribute slabs across as many machines as possible.

Using a fixed size throughout the cluster simplifies sl placement and eviction algorithms and their analyses.

Each slab starts in the *unmapped* state. INFINISW monitors the page activity rates of each slab using an e ponentially weighted moving average (EWMA) with o second period:

$$A_{\text{current}}(s) = \alpha\, A_{\text{measured}}(s) + (1-\alpha)\, A_{\text{old}}(s)$$

where α is the smoothing factor ($\alpha = 0.2$ by defau and $A(s)$ refers to total page-in and page-out activities for slab s (initialized to zero).

When $A_{\text{current}}(s)$ crosses a threshold (HotSlab), INFINISWAP initiates remote placement (§4.2) to map the slab to a remote machine's memory. This late binding helps INFINISWAP avoid unnecessary slab mapping and potential memory inefficiency. We set HotSlab to 20 page I/O requests/second. In our current design, pages are not proactively moved to remote memory. Instead, they are written to remote memory via RDMA WRITE on subsequent page-out operations.

To keep track of whether a page can be found in remote memory, INFINISWAP maintains a bitmap of all pages. All bits are initialized to zero. After a page is written out to remote memory, its corresponding bit is set. Upon failure of a remote machine where a slab is mapped or when a slab is evicted by the remote INFINISWAP daemon, all the bits pertaining to that slab are reset.

In addition to being evicted by the remote machine or due to remote failure, INFINISWAP block devices may preemptively remove a slab from remote memory if $A_{\text{current}}(s)$ goes below a threshold (ColdSlab). Our current implementation does not use this optimization.

4.2 Remote Slab Placement

When the paging activity of an unmapped slab crosses the HotSlab threshold, INFINISWAP attempts to map that slab to a remote machine's memory.

The slab placement algorithm has multiple goals. First, it must *distribute slabs from the same block device* across as many remote machines as possible in order

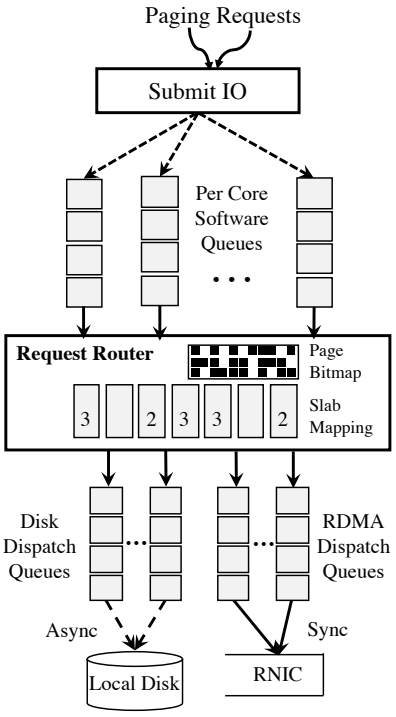


Figure 5: INFINISWAP block device overview. Each machine uses one block device as its swap partition.

to minimize the impacts of future evictions from (failures of) remote machines. Second, it attempts to *balance memory utilization* across all the machines to minimize the probability of future evictions. Finally, it must be *decentralized* to provide low-latency mapping without central coordination.

One can select an INFINISWAP daemon uniformly randomly without central coordination. However, this is known to cause load imbalance [62, 68].

Instead, we leverage power of two choices [62] to minimize memory imbalance across machines. First, INFINISWAP divides all the machines ($\mathbb{M}$) into two sets: those who already have any slab of this block device ($\mathbb{M}_{\text{old}}$) and those who do not ($\mathbb{M}_{\text{new}}$). Next, it contacts two INFINISWAP daemons and selects the one with the lowest memory usage. It first selects from $\mathbb{M}_{\text{new}}$ and then, if required, from $\mathbb{M}_{\text{old}}$. The two-step combination distributes slabs across many machines while decreasing load imbalance in a decentralized manner.

4.3 I/O Pipelines

The VMM submits page write and read requests to INFINISWAP block device using the block I/O interface (Figure 5). We use the multi-queue block IO queuing mechanism [15] in INFINISWAP. Each CPU core is configured with an individual software staging queue, where block (page) requests are staged. The request router consults the slab mapping and the page bitmap to determine how to forward them to disk and/or remote memory.

Each RDMA dispatch queue has a limited number of entries, each of which has a registered buffer for RDMA communication. Although the number of RDMA dispatch queues is the same as that of CPU cores, they do not follow one-to-one mapping. Each request from a core is assigned to a random RDMA dispatch queue by hashing its address parameter to avoid load imbalance.

Page Writes. For a page write, if the corresponding slab is mapped, a write request is duplicated and put into both RDMA and disk dispatch queues. The content of the page is copied into the buffer of RDMA dispatch entry, and the buffer is shared between the duplicated requests. Once the RDMA WRITE operation completes, the page write is completed and its corresponding physical memory can be reclaimed by the kernel without waiting for the disk write. However, the RDMA dispatch entry – and its buffer – will not be released until the completion of the disk write operation. When INFINISWAP cannot get a free entry from all RDMA dispatch queues, it blocks until one is released.

For unmapped slabs, a write request is only put into the disk dispatch queue; in this case, INFINISWAP blocks until the completion of the write operation.

Page Reads. For a page read, if the corresponding slab is mapped and the page bitmap is set, an RDMA READ operation is put into the RDMA dispatch queue. When the READ completes, INFINISWAP responds back. Otherwise, INFINISWAP reads it from the disk.

Multi-Page Requests. To optimize I/O requests, the VMM often batches multiple page requests together and sends one multi-page (batched) request. The maximum batch size in the current implementation of INFINISWAP is 128 KB (i.e., 32 4 KB pages). The challenge in handling multi-page requests arises in cases where pages cross slab boundaries, especially when some slabs are mapped and others are unmapped. In these cases, INFINISWAP waits until operations on all the pages in that batch have completed in different sources; then, it completes the multi-page I/O request.

4.4 Handling Slab Evictions

The decision to evict a slab (§5.2) is communicated to a block device via the *EVICT* message from the corresponding INFINISWAP daemon. Upon receiving this message, the block device marks the slab as unmapped and resets the corresponding portion of the bitmap. All future requests will go to disk.

Next, it waits for all the in-flight requests in the corresponding RDMA dispatch queue(s) to complete, polling every 10 microseconds. Once everything is settled, INFINISWAP responds back with a *DONE* message.

Note that if $A(s)$ is above the HotSlab threshold, INFINISWAP will start remapping the slab right away. Otherwise, it will wait until $A(s)$ crosses HotSlab again.

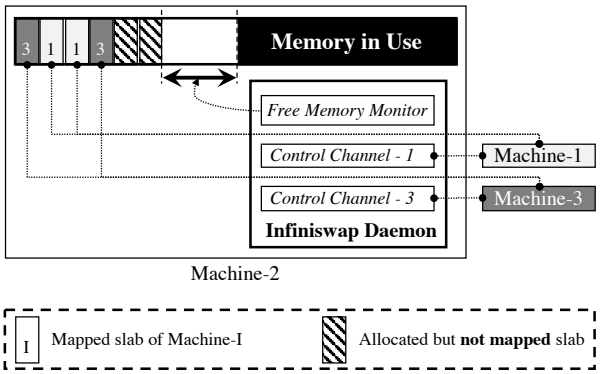


Figure 6: INFINISWAP daemon periodically monitors the available free memory to pre-allocate slabs and to perform fast evictions. Each machine runs one daemon.

4.5 Handling Remote Failures

INFINISWAP uses reliable connections for all communication and considers unreachability of remote INFINISWAP daemons (e.g., due to machine failure, daemon process crashes, etc.) as the primary failure scenario. Upon detecting a failure, the workflow is similar to that of eviction: the block device marks the slab(s) on that machine as unmapped and resets the corresponding portion(s) of the bitmap.

The key difference and a possible concern is handling in-flight requests, especially *read-after-write* scenarios. In such a case, the remote machine fails after a page (P) has been written to remote memory but before it is written to disk (i.e., P is still in the disk dispatch queue). If the VMM attempts to page P in, the bitmap will point to disk, and a disk read request will be added to the disk dispatch queue. Because all I/O requests for the same slab go to the same disk dispatch queue, such read requests will be served by the on-disk data written by the previous write operation.

In the current implementation, INFINISWAP does not handle transient failures separately. A possible optimization would be to use a timeout before marking the corresponding slabs unmapped.

5 Transparent Remote Memory Reclamation via INFINISWAP Daemon

In this section, we describe how INFINISWAP daemons (Figure 6) monitor and manage memory (§5.1) and perform slab evictions to minimize remote and local performance impacts (§5.2).

5.1 Memory Management

The core functionality of each INFINISWAP daemon is to claim memory on behalf of remote INFINISWAP block devices as well as reclaiming them on behalf of local applications. To achieve this, the daemon monitors the total

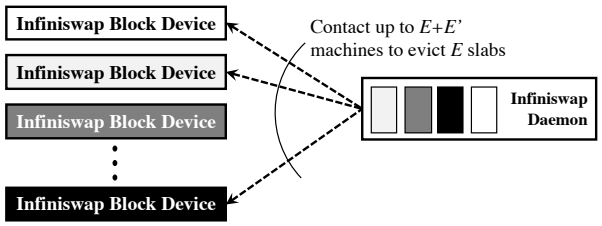


Figure 7: INFINISWAP daemon employs batch eviction (i.e., contacting E' more slabs to evict E slabs) for fast eviction of E lightly active slabs.

memory usage of everything else running on the machine using an EWMA with one second period:

$$U_{\text{current}} = \beta\, U_{\text{measured}} + (1-\beta)\, U_{\text{old}}$$

where β is the smoothing factor ($\beta = 0.2$ by default) and U refers to total memory usage (initialized to 0).

Given the total memory usage, INFINISWAP daemon focuses on maintaining a HeadRoom amount of free memory in the machine by controlling its own total memory allocation at that point. The optimal value of HeadRoom should be dynamically determined based on the amount of memory and the applications running in each machine. Our current implementation does not include this optimization and uses 8 GB HeadRoom by default on 64 GB machines.

When the amount of free memory grows above HeadRoom, INFINISWAP proactively allocates slabs of size SlabSize and marks them as unmapped. Proactive allocation of slabs makes the initialization process faster when an INFINISWAP block device attempts to map to that slab; the slab is marked mapped at that point.

5.2 Decentralized Slab Eviction

When free memory shrinks below HeadRoom, INFINISWAP daemon proactively releases slabs in two stages. It starts by releasing unmapped slabs. Then, if necessary, it *evicts* E mapped slabs as described below.

Because applications running on the local machine do not care which slabs are evicted, when INFINISWAP must evict, it focuses on minimizing the performance impact on the machines that are remotely paging. The key challenge arises from the fact that remote INFINISWAP block devices directly interact with their allocated slab(s) via RDMA READ/WRITE operations without any involvement of INFINISWAP daemons. While this avoids CPU involvements, it also prevents INFINISWAP from making any educated guess about performance impact of evicting one or more slabs without first communicating with the corresponding block devices.

This problem can be stated formally as follows. *Given S mapped slabs, how to release the E least-active ones to leave more than* HeadRoom *free memory?*

At one extreme, the solution is simple with global

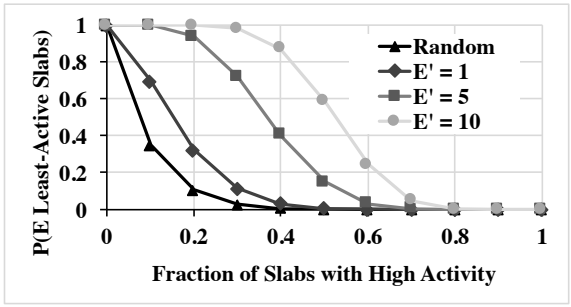


Figure 8: Analytical eviction performance for evicting $E(=10)$ slabs for varying values of E'. Random refers to evicting E slabs one-by-one uniformly randomly.

knowledge. INFINISWAP daemon can contact *all* block devices in the cluster to determine the least-used E slabs and evict them. This is prohibitive when E is significantly smaller than the total number of slabs in the cluster. Having a centralized controller would not have helped either, because this would require all INFINISWAP block devices to frequently report their slab activities.

At the other extreme, one can randomly pick one slab at a time without any communication. However, in this case, the likelihood of evicting a busy slab is very high. Consider a parameter $p_b \in [0,1]$, and assume that a slab is busy (i.e., it is experiencing paging activities beyond a fixed threshold) with probability p_b. If we now reformulate the problem to *finding E lightly active slabs* instead of the least-active ones, the probability would be $(1-p_b)^E$. As the cluster becomes busier (p_b increases), this probability plummets (Figure 8).

Batch Eviction. Instead of randomly evicting slabs without any communication, we perform bounded communication to leverage generalized power of choices [68]. Similar techniques had been used before for task scheduling and input selection [67, 75].

For E slabs to evict, INFINISWAP daemon considers $E+E'$ slabs, where $E' \leq E$. Upon communicating with the machines hosting those $E+E'$ slabs, it evicts E least-active ones (i.e., the E slabs of $E+E'$ with the lowest $A(.)$ values). The probability of finding E lightly active slabs in this case is $\sum_{E}^{E+E'} (1-p_b)^i p_b^{E+E'-i} \binom{E+E'}{i}$.

Figure 8 plots the effectiveness of batch eviction for different values of E' for $E = 10$. Even for moderate cluster load, the probability of evicting lightly active slabs are significantly higher using batch eviction.

The actual act of eviction is initiated when the daemon sends *EVICT* messages to corresponding block devices. Once a block device completes necessary bookkeeping (§4.4), it responds with a *DONE* message. Only then INFINISWAP daemon releases the slab.

6 Implementation

INFINISWAP is a virtual block device that can be used as a swap partition, for example, `/dev/infiniswap0`.

We have implemented INFINISWAP as a loadable kernel module for Linux 3.13.0 and beyond in about 3500 lines of C code. Our block device implementation is based on nbdX, a network block device over Accelio framework, developed by Mellanox[2]. We also rely on stackbd [21] to redirect page I/O requests to the disk to handle possible remote failures and evictions. INFINISWAP daemons are implemented and run as user-space programs.

Control Messages. INFINISWAP components use message passing to transfer memory information and memory service agreements. There are eight message types. Four of them are used for placement and the rest (e.g., *EVICT*, *DONE*) are used for eviction.

A detailed list of messages, along with how they are used during placement and eviction, and corresponding sequence diagrams can be found in Appendix B.1.

Connection Management. INFINISWAP uses reliable connections for all communications. It uses one-sided RDMA READ/WRITE operations for data plane activities; both types of messages are posted by the block device. All control plane messages are transferred using RDMA SEND/RECV operations.

INFINISWAP daemon maintains individual connections for each block device connected to it instead of one for each slab. Similarly, INFINISWAP block devices maintain one connection for each daemon instead of per-slab connections. Overall, for each active block device-daemon pair, there is one RDMA connection shared between the data plane and the control plane.

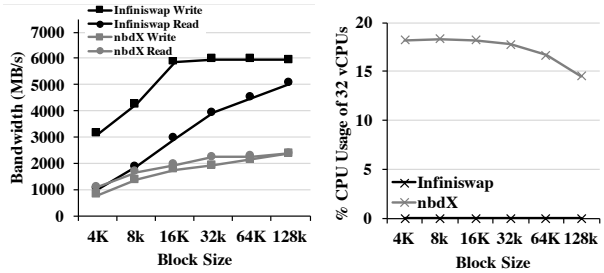
7 Evaluation

We evaluated INFINISWAP on a 32-machine, 56 Gbps Infiniband cluster on CloudLab [5] and highlight the results as follows:

- INFINISWAP provides 2×–4× higher I/O bandwidth than Mellanox nbdX [2]. While nbdX saturates 6 remote virtual cores, INFINISWAP uses none (§7.1).
- INFINISWAP improves throughputs of unmodified VoltDB, Memcached, PowerGraph, GraphX, and Apache Spark by up to 4× (0.94×) to 15.4× (7.8×) over disk (nbdX) and tail latencies by up to 61× (2.3×) (§7.2).
- INFINISWAP ensures fast recovery from remote failures and evictions with little impact on applications; it does not impact remote applications either (§7.3).
- INFINISWAP benefits hold in a distributed setting; it increases cluster memory utilization by 1.47× using a small amount of network bandwidth (§7.4).

Experimental Setup. Unless otherwise specified, we use SlabSize = 1 GB, HeadRoom = 8 GB, HotSlab = 20 paging activities per second, and $\alpha = \beta = 0.2$ in all the experiments. For comparison, nbdX also utilizes remote memory for storing data.

Each of the 32 machines had 32 virtual cores and 64 GB of physical memory.



(a) Bandwidth **(b)** Remote CPU Usage

Figure 9: INFINISWAP provides higher read and write bandwidths without remote CPU usage, whereas Mellanox nbdX suffers from high CPU overheads and lower bandwidth.

7.1 INFINISWAP Performance as a Block Device

Before focusing on INFINISWAP' effectiveness as a decentralized remote paging system, we focus on its raw performance as a block device. We compare it against nbdX and do not include disk because of its significantly lower performance. We used `fio` [6] – a well-known disk benchmarking tool – for these benchmarks.

For both INFINISWAP and nbdX, we performed parameter sweeps by varying the number of threads in `fio` from 1 to 32 and I/O depth from 2 to 64. Figure 9a shows the highest average bandwidth observed for different block sizes across all these parameter combinations for both block devices. In terms of bandwidth, INFINISWAP performs between 2× and 4× better than nbdX and saturates the 56 Gbps network at larger block sizes.

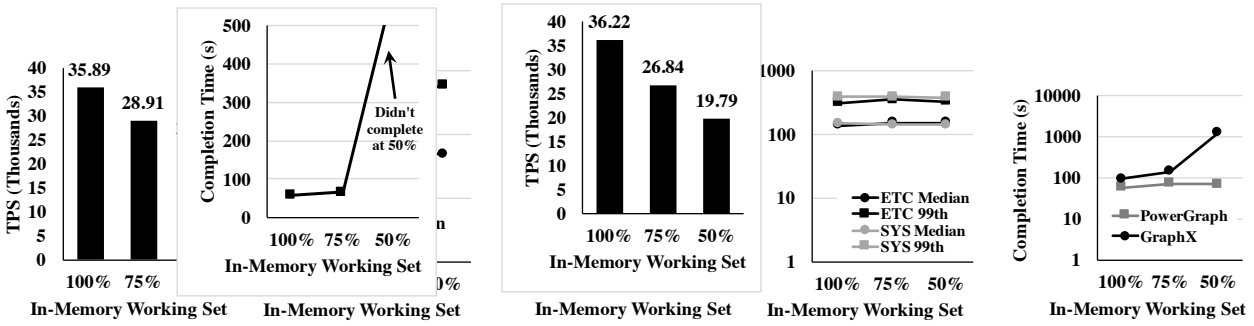
More importantly, we observe that due to nbdX's involvement in copying data to and from RAMdisk at the remote side, it has excessive CPU overheads (Figure 9b) and becomes CPU-bound for smaller block sizes. It often saturates the 6 virtual cores it runs on. In contrast, INFINISWAP bypasses remote CPU in the data plane and has close to zero CPU overheads in the remote machine.

7.2 INFINISWAP's Impact on Applications

In this section, we focus on INFINISWAP's performance on multiple memory-intensive applications with a variety of workloads (Figure 10) and compare it to that of disk (Figure 1) and nbdX (Figure 11).

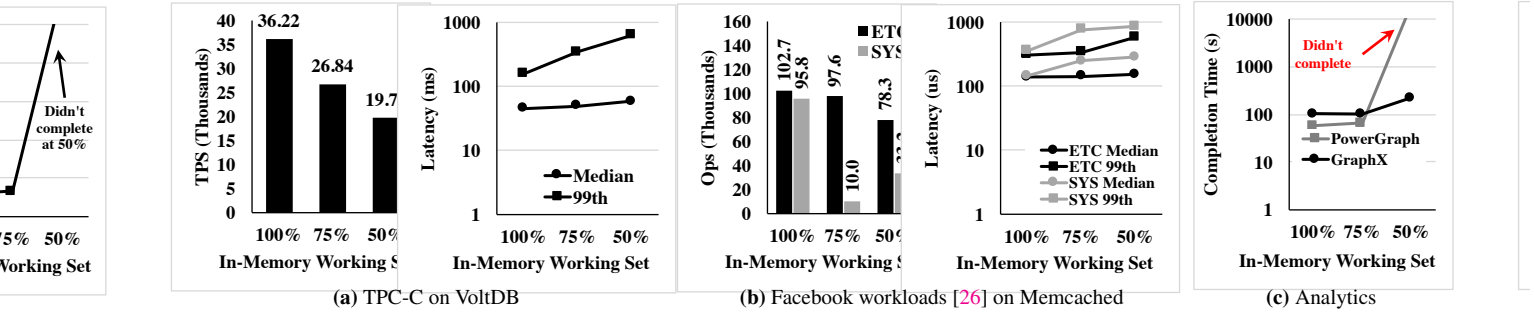
Workloads. We used four memory-intensive application and workload combinations:

1. TPC-C benchmark [22] on VoltDB [23];
2. Facebook workloads [26] on Memcached [18];
3. Twitter graph [52] on PowerGraph [45]; and
4. Twitter data on GraphX [46] and Apache Spark [77].



(a) TPC-C on VoltDB **(b)** Facebook workloads [26] on Memcached **(c)** Analytics

Figure 10: INFINISWAP performance for the same applications in Figure 1 for the same configurations. All plots show single-machine, server-side performance, and the median value of five runs. Lower is better for the latency-related plots (lines) and the opposite is true for the throughout-related ones (bars). Note the logarithmic Y-axes in the latency/completion time plots.



(a) TPC-C on VoltDB **(b)** Facebook workloads [26] on Memcached **(c)** Analytics

Figure 11: nbdX performance for comparison. All plots show single-machine, server-side performance, and the median value of five runs. Lower is better for the latency-related plots (lines) and the opposite is true for the throughout-related ones (bars). Note the logarithmic Y-axes in the latency/completion time plots.

Methodology. We focused on single-machine performance and considered three configurations – 100%, 75%, and 50% – for each application. We started with the 100% configuration by creating an `lxc` container with large enough memory to fit the entire workload in memory. We measured the peak memory usage, and then ran 75% and 50% configurations by creating containers with enough memory to fit those fractions of the peak usage. For INFINISWAP and nbdX, we use a single remote machine as the remote swap space.

7.2.1 VoltDB

VoltDB is an in-memory, transactional database that can import, operate on, and export large amounts of data at high speed, providing ACID reliability and scalability. We use its community version available on Github.

We use TPC-C to create transactional workloads on VoltDB. TPC-C performs 5 different types of transactions either executed on-line or queued for deferred execution to simulate an order-entry environment. We set 256 warehouses and 8 sites in VoltDB to achieve a reasonable single-container workload of 11.5 GB and run 2 million transactions.

We observe in Figure 10a that, unlike disk (Figure 1a), performance using INFINISWAP drops linearly instead of super-linearly when smaller amounts of workloads fit in memory. Using INFINISWAP, VoltDB experiences only a 1.5× reduction in throughput instead of 24× using disk in the 50% case. In particular, INFINISWAP improves VoltDB throughput by 15.4× and 99th-percentile latency by 19.7× in comparison to paging to disk. nbdX's performance is similar to that of INFINISWAP (Figure 11a).

Overheads of Paging. To understand why INFINISWAP's performance drops significantly when the workload does not fit into memory even though it is never paging to disk, we analyzed and compared its CPU and memory usage with all other considered applications (see Appendix A). We believe that because VoltDB is more CPU-intensive than most other memory-intensive workloads we considered, the overheads of paging (e.g., context switches) have a larger impact on its performance.

We note that paging-aware data structure placement (by modifying VoltDB) can help in mitigating this issue [36, 74]. We consider this a possible area of future work.

7.2.2 Memcached

Memcached is an in-memory object caching system that provides a simple key-value interface.

We use `memaslap`, a load generation and benchmarking tool for Memcached, to measure performance using recent data published by Facebook [26]. We pick ETC and SYS to explore the performance of INFINISWAP on

workloads with different rates of SET operations. Our experiments start with an initial phase, where we use 10 million SET operations to populate a Memcached server. We then perform another set of 10 million queries in the second phase to simulate the behavior of a given workload. ETC has 5% SETs and 95% GETs. The key size is fixed at 16 bytes and 90% of the values are evenly distributed between 16 and 512 bytes [26]. The workload size is measured to be around 9 GB. SYS, on the other hand, is SET-heavy, with 25% SET and 75% GET operations. 40% of the keys have length from 16 to 20 bytes, and the rest range from 20 to 45 bytes. Values of size between 320 and 500 bytes take up 80% of the entire data, 8% of them are smaller, and 12% sit between 500 and 10000 bytes. The workload size is measured to be 14.5 GB. We set the memory limit in Memcached configurations to ensure that for 75% and 50% configurations it will respond using the swap space.

First, we observe in Figure 10b that, unlike disk (Figure 1b), performance using INFINISWAP remains steady instead of facing linear or super-linear drops when smaller amounts of workloads fit in memory. Using INFINISWAP, Memcached experiences only 1.03× (1.3×) reduction in throughput instead of 4× (17.4×) using disk for the 50% case for the GET-dominated ETC (SET-heavy SYS) workload. In particular, INFINISWAP improves Memcached throughput by 4.08× (15.1×) and 99th-percentile latency by 36.3× (61.4×) in comparison to paging to disk.

Second, nbdX does not perform as well as it does for VoltDB. Using nbdX, Memcached experiences 1.3× (3×) throughput reduction for the 50% case for the GET-dominated ETC (SET-heavy SYS) workload. INFINISWAP improves Memcached throughput by 1.24× (2.45×) and 99th-percentile latency by 1.8× (2.29×) in comparison to paging to nbdX. nbdX's performance is not very stable either (Figure 11b).

Pitfalls of Remote CPU Usage by nbdX. When the application itself is not CPU-intensive, the differences between INFINISWAP and nbdX designs become clearer. As paging activities increase (i.e., for the SYS workload), nbdX becomes CPU-bound in the remote machine; its performance drops and becomes unpredictable.

7.2.3 PowerGraph

PowerGraph is a framework for large-scale machine learning and graph computation. It provides parallel computation on large-scale natural graphs, which usually have highly skewed power-law degree distributions.

We run TunkRank [1], an algorithm to measure the influence of a Twitter user based on the number of that user's followers, on PowerGraph. TunkRank's implementation on PowerGraph was obtained from [9]. We use a Twitter dataset of 11 million vertices as the input. The

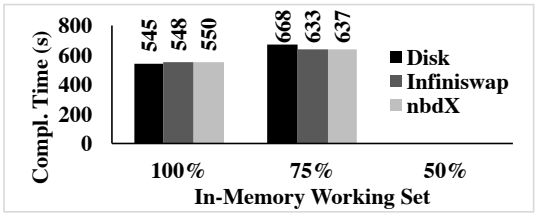


Figure 12: Comparative performance for PageRank using Apache Spark. The 50% configuration fails for all alternatives because Spark starts thrashing.

dataset size is 1.3 GB. We use the asynchronous engine of PowerGraph and `tsv` input format with the number of CPU cores set to 2, resulting in a 9 GB workload.

Figure 10c shows that, unlike disk (Figure 1c), performance using INFINISWAP remains stable. Using INFINISWAP, PowerGraph experiences only 1.24× higher completion time instead of 8× using disk in the 50% case. In particular, INFINISWAP improves PowerGraph's completion by 6.5× in comparison to paging to disk.

nbdX did not even complete at 50% (Figure 11c).

7.2.4 GraphX and Apache Spark

GraphX is a specialized graph processing system built on top of the Apache Spark in-memory analytics engine. We used Apache Spark 2.0.0 to run PageRank on the same Twitter user graph using both GraphX and vanilla Spark. For the same workload, GraphX could run using 12 GB maximum heap, but Spark needed 16 GB.

Figure 10c shows that INFINISWAP makes a 2× performance improvement over the case of paging to disk for the 50% configuration for GraphX. However, for Spark, all three of them fail to complete for the 50% configuration (Figure 12). In both cases, the underlying engine (i.e., Spark) starts thrashing – applications oscillate between paging out and paging in making little or no progress. In general, GraphX has smaller completion times than Spark for our workload.

7.3 Performance of INFINISWAP Components

So far we have considered INFINISWAP's performance without any eviction or failures. In this section, we analyze from both block device and daemon perspectives.

7.3.1 INFINISWAP Block Device

For these experiments, we present results for the 75% VoltDB configuration. We select VoltDB because it experienced one of the lowest performance benefits using INFINISWAP. We run it in one machine and distribute its slabs across 6 machines' remote memory. We then introduce different failure and eviction events to measure VoltDB's throughput loss (Figure 13).

Handling Remote Failures. First, we randomly turned off one of the 6 machines in 10 different runs; the failed (turned-off) machine did not join back. The average

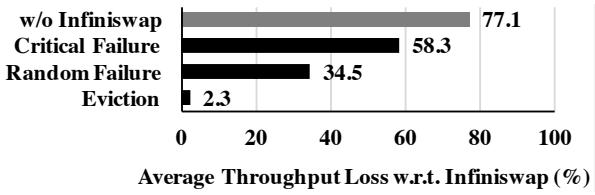


Figure 13: Average throughput loss of VoltDB with 75% in-memory working set w.r.t. INFINISWAP from different failure a

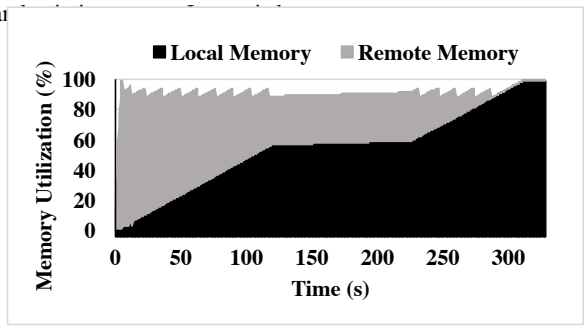


Figure 14: INFINISWAP daemon proactively evicts slabs to ensure that a local Memcached server runs smoothly. The white/empty region toward the top represents HeadRoom.

throughput loss was about 34.5% in comparison to INFINISWAP without any failures. However, we observed that the timing of failure has a large impact (e.g., during high paging activity or not). So, to create an adversarial scenario, we turned off one of the 6 machines again, but in this case, we failed the highest-activity machine during its peak activity period. The average throughout loss increased to 58.3%.

Handling Evictions. In this experiment, we evicted 1 slab from one of the remote machines every second and measured the average throughput loss to be about 2.3%, on average, for each eviction event (of 7–29 eviction-and-remapping events). As before, eviction of a high-activity slab had a slightly larger impact than that of one with lower activity.

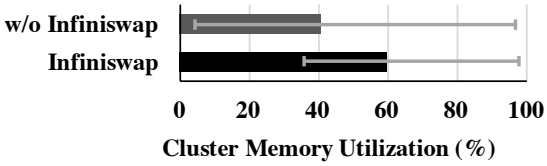
Time to Map a Slab. We also measured the time INFINISWAP takes to map (for the first time) or remap (due to eviction or failure) a slab. The median time was 54.25 milliseconds, out of which 53.99 went to Infiniband memory registration. Memory registration is essential and incurs the most interfering overhead in Infiniband communication [59]; it includes address translation, and pinning pages in memory to prevent swapping. Note that preallocation of slabs by INFINISWAP daemons mask close to 400 milliseconds, which would otherwise have been added on top of the 54.25 milliseconds. Detailed breakdown is given in Appendix B.2.

7.3.2 INFINISWAP Daemon

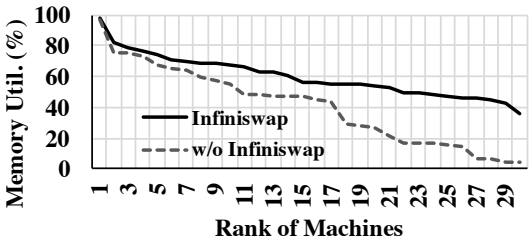
Now, we focus on INFINISWAP daemon's reaction time to increase in memory demands of local applications. For this experiment, we set HeadRoom to be 1 GB and

	ETC		SYS	
	W/o	With	W/o	With
Ops (Thousands)	95.9	94.1	96.0	93.5
Median Latency (us)	152.0	152.0	152.0	156.0
99th Latency (us)	319.0	318.0	327.0	343.0

Table 1: Performance of an in-memory Memcached server with and without INFINISWAP using remote memory.



(a) Cluster memory utilization



(b) Memory utilization of individual machines

Figure 15: Using INFINISWAP, memory utilization increases and memory imbalance decreases significantly. Error bars in (a) show the maximum and the minimum across machines.

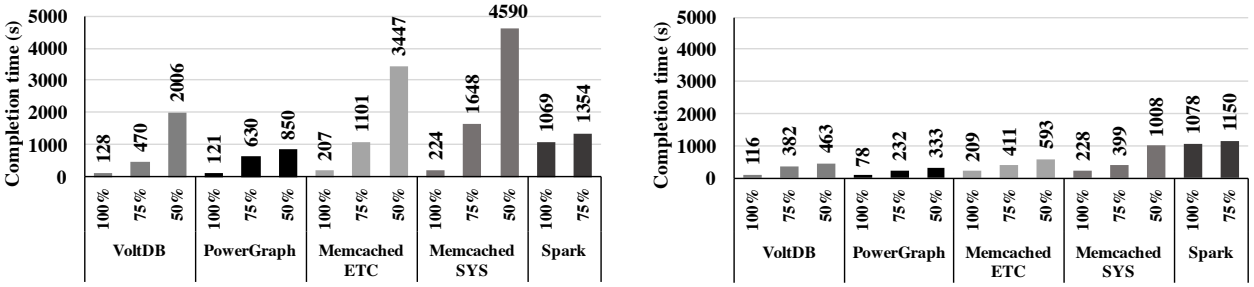
started at time zero with INFINISWAP hosting a large number of remote slabs. We started a Memcached server soon after and started performing the ETC workload. Figure 14 shows how INFINISWAP daemon monitored local memory usage and proactively evicted remote slabs to make room – the white/empty region toward the top represents the HeadRoom distance INFINISWAP strived to maintain.

After about 120 seconds, when Memcached stopped allocating memory for a while, INFINISWAP stopped retreating as well. INFINISWAP resumed slab evictions when Memcached's allocation started growing again.

To understand whether INFINISWAP retreated fast enough not to have any impact on Memcached performance, we measured its throughput as well as median and 99th-percentile latencies (Table 1), observing less than 2% throughput loss and at most 4% increase in tail latency. Results for the SYS workload were similar.

Time to Evict a Slab The median time to evict a slab was 363 microseconds. A detailed breakdown of events is provided in Appendix B.2.

The eviction speed of INFINISWAP daemon can keep up with the rate of memory allocation in most cases. In extreme cases, the impact on application performance can be reduced by adjusting HeadRoom.



(a) Without INFINISWAP **(b)** INFINISWAP

Figure 16: Median completion times of containers for different configurations in the cluster experiment. INFINISWAP's benefits translate well to a larger scale in the presence of high application concurrency.

7.4 Cluster-Wide Performance

So far we have considered INFINISWAP's performance for individual applications and analyzed its components. In this section, we deploy INFINISWAP on a 32-machine RDMA cluster and observe whether these benefits hold in the presence of concurrency and at scale.

Methodology. For this experiment, we used the same applications, workloads, and configurations from Section 7.2 to create about 90 containers. We created an equal number of containers for each application-workload combination. About 50% of them were using the 100% configuration, close to 30% used the 75% configuration, and the rest used the 50% configuration.

We placed these containers randomly across 32 machines to create an memory imbalance scenario similar to those shown in Figure 2 and started all the containers at the same time. We measured completion times for the workload running each container; for VoltDB and Memcached completion time translates to transactions- or operations-per-second.

7.4.1 Cluster Utilization

Figure 15a shows that INFINISWAP increased total cluster memory utilization by 1.47× by increasing it to 60% on average from 40.8%. Moreover, INFINISWAP significantly decreased memory imbalance (Figure 15b): the maximum-to-median utilization ratio decreased from 2.36× to 1.6× and the maximum-to-minimum utilization ratio decreased from 22.5× to 2.7×.

Increase in Network Utilization. We also measured the total amount of network traffic over RDMA in the case of INFINISWAP. This amounted to less than 1.88 TB over 1300 seconds across 32 machines or 380 Mbps on average for each machine, which is less than 1% of each machine's 56 Gbps interface.

7.4.2 Application-Level Performance

Finally, Figure 16 shows the overall performance of INFINISWAP. We observe that INFINISWAP's benefits are not restricted only to microbenchmarks, and it works well in the presence of cluster dynamics of many applications. Although improvements are sometimes lower than those observed in controlled microbenchmarks, INFINISWAP still provides 3×–6× improvements for the 50% configurations.

8 Discussion and Future Work

Slab Size. For simplicity and efficiency, unlike some remote paging systems [38], INFINISWAP uses moderately large slabs (SlabSize), instead of individual pages, for remote memory management. This reduces INFINISWAP's meta-data management overhead. However, too large a SlabSize can lower flexibility and decrease space efficiency of remote memory. Selecting the optimal slab size to find a good balance between management overhead and memory efficiency is part of our future work.

Application-Aware Design. Although application transparency in INFINISWAP provides many benefits, it limits INFINISWAP's performance for certain applications. For example, database applications have hot and cold tables, and adapting to their memory access patterns can bring considerable performance benefits [74]. It may even be possible to automatically infer memory access patterns to gain significant performance benefits [36].

OS-Aware Design. Relying on swapping allows INFINISWAP to provide remote memory access without OS modifications. However, swapping introduces unavoidable overheads, such as context switching. Furthermore, the amount of swapped data can vary significantly over time and across workloads even for the same application. Currently, INFINISWAP cannot provide predictable performance without any controllable swap mechanism inside the OS. We would like to explore what can be done if we are allowed to modify OS-level decisions, such

as changing its memory allocator or not making context switches due to swapping.

Application Differentiation. Currently, INFINISWAP provides remote memory to all the applications running on the machine. It cannot distinguish between pages from specific applications. Also, there are no limitations in remote memory usage for each application. Being able to differentiate the source of a page will allow us to manage resources better and isolate applications.

Network Bottleneck. INFINISWAP assumes that it does not have to compete with other applications for the RDMA network; i.e., the network is not a bottleneck. However, as the number of applications using RDMA increases, contentions will increase as well. Addressing this problem requires mechanisms to provide isolation in the network among competing applications.

9 Related Work

Resource Disaggregation. To decouple resource scaling and to increase datacenter efficiency, resource disaggregation and rack-scale computing have received significant attention in recent years, with memory disaggregation being the primary focus [10–12, 33, 49, 56, 57]. Recent feasibility studies [42, 53, 70] have shown that memory disaggregation may indeed be feasible even at a large scale, modulo RDMA deployment at datacenter-scale [48, 79]. INFINISWAP realizes this vision in practice and exposes the benefits of memory disaggregation to *any* user application without modifications.

Remote Memory Paging. Paging out to remote memory instead of local disks is a known idea [25, 31, 37, 39–41, 58, 64]. However, their performance and promises were often limited by slow networks and high CPU overheads. Moreover, they rely on central coordination for remote server selection, eviction, and load balancing. INFINISWAP focuses on a decentralized solution for the RDMA environment.

HPBD [55] and Mellanox nbdX [2] come the closest to INFINISWAP. Both of them can be considered as network-attached-storage (NAS) systems that use RAMdisk on the server side and are deployed over RDMA networks. However, there are several major differences that make INFINISWAP more efficient, resilient, and load balanced. First, they rely on remote RAMdisks, and data copies to and from RAMdisks become CPU-bound; in contrast, INFINISWAP does not involve remote CPUs, which increases efficiency. Second, they do not perform dynamic memory management, ignoring possibilities of evictions and subsequent issues. Finally, they do not consider fault tolerance nor do they attempt to minimize the impact of failures.

Software Distributed Shared Memory (DSM). DSM systems [29, 54, 65] expose a shared global address space to user applications. Traditionally, these systems have suffered from communication overheads to maintain coherence. To avoid coherence costs, the HPC community has favored the Partitioned Global Address Space (PGAS) model [30, 34] instead. However, PGAS systems require complete rewriting of user applications with explicit awareness of remote data accesses. With the advent of RDMA, there has been a renewed interest in DSM research, especially via the key-value interface [35, 51, 60, 63, 66, 69]. However, most of these solutions are either limited by their interface or require careful rethinking/rewriting of user applications. INFINISWAP, on the contrary, is a transparent, efficient, and scalable solution that opportunistically leverages remote memory.

10 Conclusion

This paper rethinks the well-known remote memory paging problem in the context of RDMA. We have presented INFINISWAP, a pragmatic solution for memory disaggregation without requiring any modifications to applications, OSes, or hardware. Because CPUs are not involved in INFINISWAP's data plane, we have proposed scalable, decentralized placement and eviction algorithms leveraging the power of many choices. Our in-depth evaluation of INFINISWAP on unmodified VoltDB, Memcached, PowerGraph, GraphX, and Apache Spark has demonstrated its advantages in substantially improving throughputs (up to 16.3×), median latencies (up to 5.5×), and tail latencies (up to 58×) over disks. It also provides benefits over existing RDMA-based remote memory paging solutions by avoiding remote CPU involvements. INFINISWAP increases the overall memory utilization of a cluster, and its benefits hold at scale.

Acknowledgments

Special thanks go to the entire CloudLab team – especially Robert Ricci, Leigh Stoller, and Gary Wong – for pooling together enough resources to make INFINISWAP experiments possible. We would also like to thank the anonymous reviewers and our shepherd, Mike Dahlin, for their insightful comments and feedback that helped improve the paper. This work was supported in part by National Science Foundation grants CCF-1629397, CNS-1563095, CNS-1617773, by the ONR grant N00014-15-1-2163, and by an Intel grant on low-latency storage systems.

References

[1] A Twitter Analog to PageRank. http://thenoisychannel.com/2009/01/13/a-twitter-analog-to-pagerank.

[2] Accelio based network block device. https://github.com/accelio/NBDX.

[3] Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing. Accessed: 2017-02-02.

[4] Apache Hadoop NextGen MapReduce (YARN). http://goo.gl/etTGA.

[5] CloudLab. https://www.cloudlab.us.

[6] Fio - Flexible I/O Tester. https://github.com/axboe/fio.

[7] Google Compute Engine Pricing. https://cloud.google.com/compute/pricing. Accessed: 2017-02-02.

[8] Google Container Engine. https://cloud.google.com/container-engine/.

[9] Graph Analytics Benchmark in CloudSuite. http://parsa.epfl.ch/cloudsuite/graph.html.

[10] HP Moonshot System: The world's first software-defined servers. http://h10032.www1.hp.com/ctg/Manual/c03728406.pdf.

[11] HP: The Machine. http://www.labs.hpe.com/research/themachine/.

[12] Intel RSA. http://www.intel.com/content/www/us/en/architecture-and-technology/rsa-demo-x264.html.

[13] Kubernetes. http://kubernetes.io.

[14] Linux memory management. http://www.tldp.org/LDP/tlk/mm/memory.html.

[15] Linux Multi-Queue Block IO Queueing Mechanism (blk-mq). https://www.thomas-krenn.com/en/wiki/Linux_Multi-Queue_Block_IO_Queueing_Mechanism_(blk-mq).

[16] Mellanox ConnectX-3 User Manual. http://www.mellanox.com/related-docs/user_manuals/ConnectX-3_VPI_Single_and_Dual_QSFP_Port_Adapter_Card_User_Manual.pdf.

[17] Mellanox SX6036G Specifications. http://www.mellanox.com/related-docs/prod_gateway_systems/PB_SX6036G.pdf.

[18] Memcached - A distributed memory object caching system. http://memcached.org.

[19] Microsoft Azure Cloud Services Pricing. https://azure.microsoft.com/en-us/pricing/details/cloud-services/. Accessed: 2017-02-02.

[20] Redis, an in-memory data structure store. http://redis.io.

[21] Stackbd: Stacking a block device. https://github.com/OrenKishon/stackbd.

[22] TPC Benchmark C (TPC-C). http://www.tpc.org/tpcc/.

[23] VoltDB. https://github.com/VoltDB/voltdb.

[24] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *NSDI*, 2013.

[25] E. A. Anderson and J. M. Neefe. An exploration of network RAM. Technical Report UCB/CSD-98-1000, EECS Department, University of California, Berkeley, Dec 1994.

[26] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[27] L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 8(3):1–154, 2013.

[28] P. Bodik, I. Menache, M. Chowdhury, P. Mani, D. Maltz, and I. Stoica. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM*, 2012.

[29] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *SOSP*, 1991.

[30] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[31] H. Chen, Y. Luo, X. Wang, B. Zhang, Y. Sun, and Z. Wang. A transparent remote paging model for virtual machines. In *International Workshop on Virtualization Technology*, 2008.

[32] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *SIGCOMM*, 2013.

[33] P. Costa, H. Ballani, K. Razavi, and I. Kash. R2C2: A network stack for rack-scale computers. In *SIGCOMM*, 2015.

[34] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. Von Eicken, and K. Yelick. Parallel programming in Split-C. In *Supercomputing*, 1993.

[35] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *NSDI*, 2014.

[36] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan. Data tiering in heterogeneous memory systems. In *EuroSys*, 2016.

[37] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets. Cashmere-VLM: Remote memory paging for software distributed shared memory. In *IPPS/SPDP*, 1999.

[38] M. J. Feeley, W. E. Morgan, E. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *SOSP*, 1995.

[39] M. J. Feeley, W. E. Morgan, E. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *ACM SIGOPS Operating Systems Review*, volume 29, pages 201–212. ACM, 1995.

[40] E. W. Felten and J. Zahorjan. Issues in the implementation of a remote memory paging system. Technical Report 91-03-09, University of Washington, Mar 1991.

[41] M. D. Flouris and E. P. Markatos. The network RamDisk: Using remote memory on heterogeneous NOWs. *Journal of Cluster Computing*, 2(4):281–293, 1999.

[42] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *OSDI*, 2016.

[43] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[44] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *EuroSys*, 2013.

[45] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[46] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[47] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

[48] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. RDMA over commodity Ethernet at scale. In *SIGCOMM*, 2016.

[49] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *HotNets*, 2013.

[50] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[51] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *SIGCOMM*, 2014.

[52] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, 2010.

[53] F. Li, S. Das, M. Syamala, and V. R. Narasayya. Accelerating relational databases by leveraging remote memory and RDMA. In *SIGMOD*, 2016.

[54] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM TOCS*, 7(4):321–359, 1989.

[55] S. Liang, R. Noronha, and D. K. Panda. Swapping to remote memory over Infiniband: An approach using a high performance network block device. In *Cluster Computing*, 2005.

[56] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ISCA*, 2009.

[57] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *HPCA*, 2012.

[58] E. P. Markatos and G. Dramitinos. Implementation of a reliable remote memory pager. In *USENIX ATC*, 1996.

[59] F. Mietke, R. Rex, R. Baumgartl, T. Mehlan, T. Hoefler, and W. Rehm. Analysis of the memory registration process in the Mellanox Infiniband software stack. In *Euro-Par*, 2006.

[60] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *USENIX ATC*, 2013.

[61] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats, et al. TIMELY: RTT-based congestion control for the datacenter. In *SIGCOMM*, 2015.

[62] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. The power of two random choices: A survey of techniques and results. *Handbook of Randomized Computing*, pages 255–312, 2001.

[63] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *USENIX ATC*, 2015.

[64] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel. Nswap: A network swapping module for Linux clusters. In *Euro-Par*, 2003.

[65] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[66] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high performance storage entirely in DRAM. *SIGOPS OSR*, 43(4), 2010.

[67] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *SOSP*, 2013.

[68] G. Park. Brief announcement: A generalization of multiple choice balls-into-bins. In *PODC*, 2011.

[69] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, 2010.

[70] P. S. Rao and G. Porter. Is memory disaggregation feasible?: A case study with Spark SQL. In *ANCS*, 2016.

[71] C. Reiss. *Understanding Memory Configurations for In-Memory Analytics*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2016.

[72] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC*, 2012.

[73] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *EuroSys*, 2013.

[74] R. Stoica and A. Ailamaki. Enabling efficient OS paging for main-memory OLTP databases. In *Ninth International Workshop on Data Management on New Hardware*, 2013.

[75] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. The power of choice in data-aware cluster scheduling. In *OSDI*, 2014.

[76] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with Borg. In *EuroSys*, 2015.

[77] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

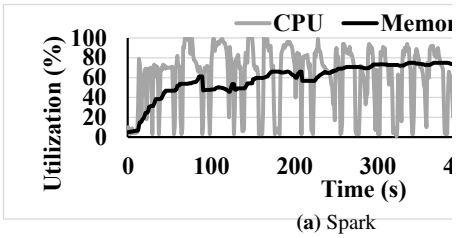
[78] Q. Zhang, M. F. Zhani, S. Zhang, Q. Zhu, R. Boutaba, and J. L. Hellerstein. Dynamic energy-aware capacity provisioning for cloud computing environments. In *ICAC*, 2012.

[79] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale RDMA deployments. In *SIGCOMM*, 2015.

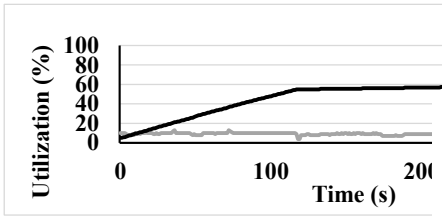
A Characteristics of the Benchmarks

We ran each benchmark application in separate containers with 16 GB memory (32 GB only for Spark) and measured their real-time CPU and memory utilizations from cold start. We make the following observations from these experiments.

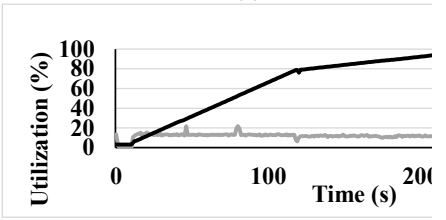
First, while memory utilizations of all applications increased gradually before plateauing, Spark has significantly higher memory utilization along with very high CPU usage (Figure 17a). This is perhaps one of the primary reasons why Spark starts thrashing when it cannot keep its working set in memory (i.e., in 75% and 50% configurations). While GraphX exhibits a similar trend (Figure 17e), its completion time is much smaller for the same workload. Even though it starts thrashing in the



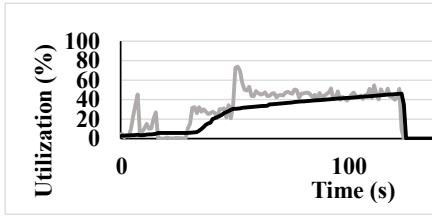
(a) Spark



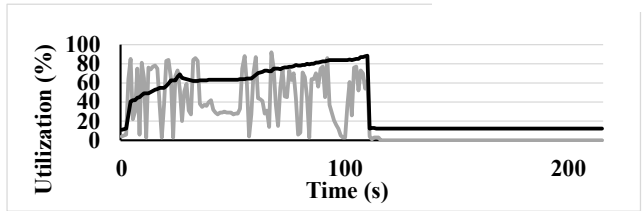
(b) Memcached-ETC



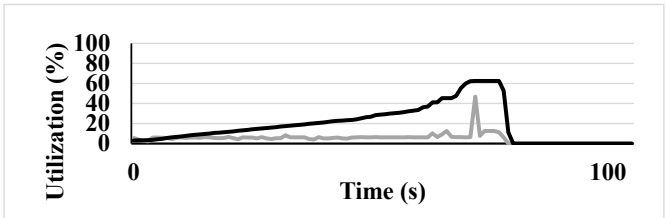
(c) Memcached-SYS



(d) VoltDB



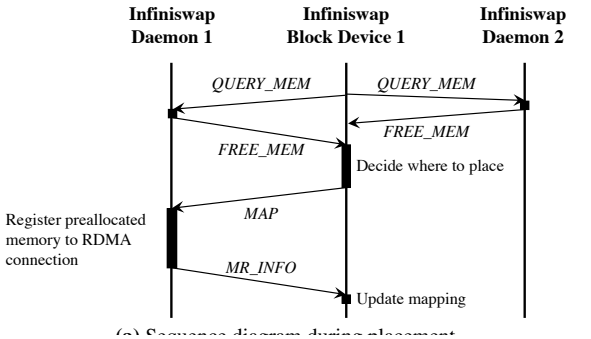
(e) GraphX



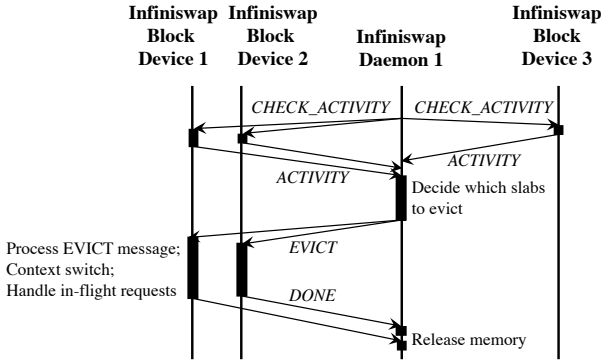
(f) PowerGraph

Figure 17: CPU and memory usage characteristics of the benchmarked applications and workloads running on containers with 16 GB memory (32 GB only for Spark). Note the increasingly smaller timescales in different X-axes due to smaller completion times of each workload.

50% configuration, it can eventually complete before spiraling out of control.



(a) Sequence diagram during placement



(b) Sequence diagram during eviction

Figure 18: Decentralized placement and eviction in INFINISWAP. (a) INFINISWAP block devices use the power of two choices to select machines with the most available memory to place each slab. (b) INFINISWAP daemons use the power of many choices to select slab(s) to evict; in this case, the daemon is contacting 3 block devices to evict 2 slabs.

Second, other than Spark and GraphX, VoltDB has at least 3× higher average CPU utilization than other benchmarks (Figure 17d). This is one posible explanation of its smaller improvements with INFINISWAP for the 50% and 75% cases in comparison to other less CPU-heavy applications – overheads of paging (e.g., context switch) was possibly a considerable fraction of VoltDB's runtimes.

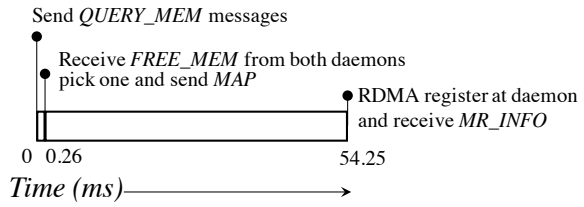
Third, both ETC and SYS workloads gradually allocate more memory over time, but ETC plateaus early because it has mostly GET operations (Figure 17b), whereas SYS keeps increasing because of its large number of SET operations (Figure 17c).

Finally, PowerGraph is the most efficient of the workloads we considered (Figure 17f). It completes faster and has the smallest resource usage footprint, both of which contribute to its consistent performances using INFINISWAP across all configurations.

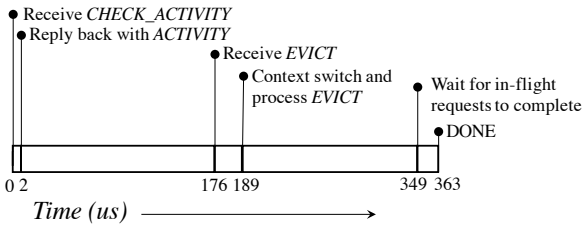
B Control Plane Details

B.1 Control Messages

INFINISWAP components use message passing to transfer memory information and memory service agreements.



(a) Timing diagram of placement



(b) Timing diagram of eviction

Figure 19: Timing diagrams (not drawn to scale) from a INFINISWAP block device's perspective during decentralized placement and eviction events.

There are eight message types; the first four of them are used by the placement algorithm (Figure 18a) and the rest are used by the eviction algorithm (Figure 18b).

1. *QUERY_MEM*: Block devices send it to get the number of available memory slabs on the daemon.
2. *FREE_MEM*: Daemons respond to *QUERY_MEM* requests with the number of available memory slabs.
3. *MAP*: Block device confirms that it has decided to use one memory slab from this daemon.
4. *MR_INFO*: Daemon sends memory registration information (`rkey`, `addr`, `len`) of an available memory slab to the block device in response to MAP.
5. *CHECK_ACTIVITY*: Daemons use this message to ask for paging activities of specific memory slab(s).
6. *ACTIVITY*: Block device's response to the *CHECK_ACTIVITY* messages.
7. *EVICT*: Daemons alert the block device which memory slab(s) it has selected to evict.
8. *DONE*: After completely redirecting the requests to the to-be-evicted memory slab(s), block device responds with this message so that the daemon can safely evict and return physical memory to its local OS.

B.2 Breakdown of Control Plane Timings

Here we breakdown the timing results from Section 7.3. Figures 19a and 19b provide details of placement and eviction timings from a INFINISWAP block device's perspective.

Parameter	Value
Datacenter OPEX	$0.04/W/month
Electricity Cost	$0.067/kWh
InfiniBand NIC Power	8.41W [16]
InfiniBand Switch Power	231W [17]
Power Usage Effectiveness (PUE)	1.1

Table 2: Cost Model Parameters [27].

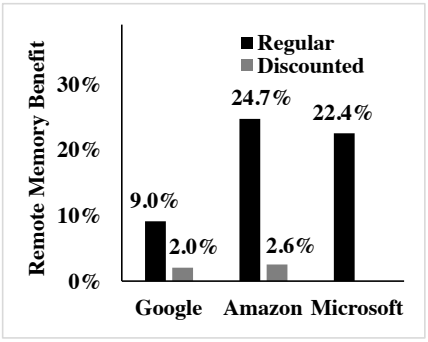


Figure 20: Revenue increases with INFINISWAP under three different cloud vendors' regular and discounted pricing models.

C Cost-Benefit Analysis

In many production clusters, memory and CPU usages are unevenly distributed across machines (§2.3) and resources are often underutilized [72, 78]. Using memory disaggregation via INFINISWAP, machines with high memory demands can use idle memory from other machines, thus enabling more applications to run simultaneously on a cluster and providing more economic benefits. Here we perform a simple cost-benefit analysis to get a better understanding of such benefits.

We limit our analysis only to RDMA-enabled clusters, and therefore, do not consider capital expenditure and depreciation cost of acquiring RDMA hardware. The major source of operational expenditure (OPEX) comes from the energy consumption of Infiniband devices – the parameters of our cost model are listed in Table 2. The average cost of INFINISWAP for a single machine is around $1.45 per month.

We also assume that there are more idle CPUs than idle memory in the cluster, and INFINISWAP's benefits are limited by the latter. For example, on average, about 40% and 30% of allocated CPUs and memory are reported to remain unused in Google clusters [72]. We use the price lists from Google Cloud Compute [7], Amazon EC2 [3], and Microsoft Azure [19] to build the benefit model. INFINISWAP is found to increase total cluster memory utilization by around 20% (7.4.1), varying slightly across different application deployment scenarios. We assume that there are 20% physical memory on each machine that has been allocated to local applications but the remainder is used as remote memory via INFINISWAP. The additional benefit is then the price of

20% physical memory after deducting the cost of operating Infiniband.

There are several price models from different vendors. In a model we call the *regular pricing model*, resource availability is strictly guaranteed. In another model from Google (preemptible instance) and Amazon (spot instance), resource can be preempted or become unavailable based on resource availability in a cluster. Of course, the resource price in the latter model is much lower than the regular model. We call it the *discounted pricing model*.

If INFINISWAP can ensure unnoticeable performance degradation to applications, remote memory can be counted under regular pricing; otherwise, discounted pricing should be used. Figure 20 shows benefits of INFINISWAP. With an ideal INFINISWAP, cluster vendors can gain up to 24.7% additional revenue. If we apply the discounted model, then it decreases to around 2%.

TuX^2: Distributed Graph Computation for Machine Learning

Wencong Xiao[†*], Jilong Xue[*◇], Youshan Miao[*], Zhen Li[†*], Cheng Chen[*],
Ming Wu[*], Wei Li[†], Lidong Zhou[*]
[†]*SKLSDE Lab, Beihang University,* [*]*Microsoft Research,* [◇]*Peking University*

Abstract

TuX^2 is a new distributed graph engine that bridges graph computation and distributed machine learning. TuX^2 inherits the benefits of an elegant graph computation model, efficient graph layout, and balanced parallelism to scale to billion-edge graphs; we extend and optimize it for distributed machine learning to support heterogeneity, a Stale Synchronous Parallel model, and a new *MEGA* (Mini-batch, Exchange, GlobalSync, and Apply) model.

We have developed a set of representative distributed machine learning algorithms in TuX^2, covering both supervised and unsupervised learning. Compared to implementations on distributed machine learning platforms, writing these algorithms in TuX^2 takes only about 25% of the code: Our graph computation model hides the detailed management of data layout, partitioning, and parallelism from developers. Our extensive evaluation of TuX^2, using large data sets with up to 64 billion edges, shows that TuX^2 outperforms state-of-the-art distributed graph engines PowerGraph and PowerLyra by an order of magnitude, while beating two state-of-the-art distributed machine learning systems by at least 48%.

1 Introduction

Distributed graph engines, such as Pregel [30], PowerGraph [17], and PowerLyra [7], embrace a *vertex-program* abstraction to express iterative computation over large-scale graphs. A graph engine effectively encodes an index of the data in a graph structure to expedite graph-traversal-based data access along edges, and supports elegant graph computation models such as Gather-Apply-Scatter (GAS) for ease of programming. A large body of research [7, 18, 22, 24, 32, 33, 36, 39, 46, 47] has been devoted to developing highly scalable and efficient graph engines through data layout, partitioning, scheduling, and balanced parallelism. It has been shown that distributed graph engines can scale to graphs with more than a trillion edges [10, 43, 38] for simple graph algorithms such as PageRank.

Early work on graph engines (e.g., GraphLab [29]) was motivated by machine learning, based on the observation that many machine learning problems can be modeled naturally and efficiently with graphs and solved by iterative convergence algorithms. However, most subsequent work on graph engines adopts a simplistic graph computation model, driven by basic graph benchmarks such as PageRank. The resulting graph engines lack flexibility and other key capabilities for efficient distributed machine learning.

We present TuX^2, a distributed graph engine for machine learning algorithms expressed in a graph model. TuX^2 preserves the benefits of graph computation, whlie also supporting the Stale Synchronous Parallel (SSP) model [20, 11, 42, 13], a heterogeneous data model, and a new *MEGA* (Mini-batch, Exchange, GlobalSync, and Apply) graph model for efficient distributed machine learning. We evaluate the performance of TuX^2 on a distributed cluster of 32 machines (with over 500 physical cores) on both synthetic and real data sets with up to 64 billion edges, using representative distributed machine learning algorithms including Matrix Factorization (MF) [16], Latent Dirichlet Allocation (LDA) [45], and Block Proximal Gradient (BlockPG) [27], covering both supervised and unsupervised learning. The graph model in TuX^2 significantly reduces the amount of code (by 73–83%) that developers need to write for the algorithms, compared to the state-of-the-art distributed machine learning platforms such as Petuum [20, 44] and Parameter Server [26]. It also enables natural graph-based optimizations such as vertex-cut for achieving balanced parallelism. Our evaluation shows that TuX^2 outperforms state-of-the-art graph engines PowerGraph and PowerLyra by more than an order of magnitude, due largely to our heterogeneous MEGA graph model. TuX^2 also beats Petuum and Parameter Server by at least 48%

thanks to a series of graph-based optimizations.

As one of our key contributions, TuX^2 bridges two largely parallel threads of research, graph computation and parameter-server-based distributed machine learning, in a unified model, advancing the state of the art in both. TuX^2 significantly expands the capabilities of graph engines in three key dimensions: data representation and data model, programming model, and execution scheduling. We propose a set of representative machine learning algorithms for evaluating graph engines on machine learning applications, guiding graph engines towards addressing the real challenges of distributed machine learning and thereby becoming more widely used in practice. We have also, through extensive evaluation on real workloads at scale, shown significant benefits in programmability, scalability, and efficiency for graph computation models in distributed machine learning.

The rest of the paper is organized as follows. §2 offers an overview of graph computation and machine learning, highlighting their connections. §3 describes TuX^2's design. §4 presents three machine learning algorithms, detailing how they are expressed and realized in TuX^2; §5 discusses the implementation and evaluation of TuX^2. We discuss related work in §6 and conclude in §7.

2 Graphs for Machine Learning

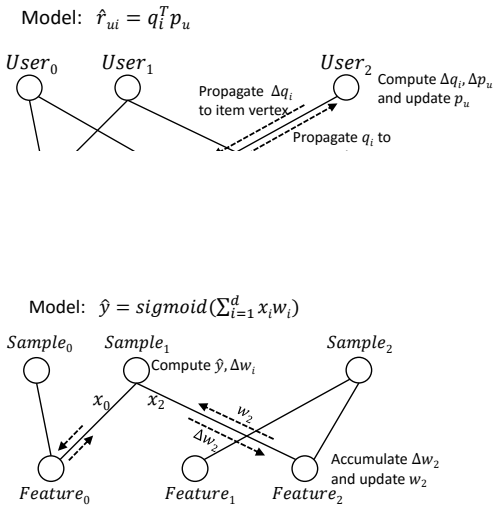
In this section, we highlight the benefits of abstraction into a graph model, show how a large class of machine learning algorithms can be mapped to graph models, and outline why existing graph engines fall short of supporting those algorithms in expressiveness and efficiency.

Graph parallel abstraction. A graph parallel abstraction models data as a graph $G = \{V, E\}$ with V the set of vertices and E the set of edges. A vertex-program P is provided to execute in parallel on each vertex $v \in V$ and interact with neighboring instances $P(u)$, where $(u, v) \in E$. The vertex-program often maintains an application-specific state associated with vertices and with edges, exchanges the state values among neighboring vertices, and computes new values during graph computation. It typically proceeds in iterations and, when a Bulk Synchronous Parallel (BSP) model is used, introduces a synchronization barrier at the end of each iteration. By constraining the interactions among nodes of a vertex-program using a graph model, this abstraction lets the underlying system encode an index of the data as a graph structure to allow fast data access along edges. Many existing state-of-the-art graph engines have adopted this parallel vertex-program approach, though the actual form of vertex-program design might vary. As a representative graph model, the GAS model proposed in PowerGraph [17] defines three phases of a vertex-program: Gather, Apply, and Scatter. For each vertex u, the *gather* phase collects information about neighbor vertices and edges of u through a generalized sum function that is commutative and associative. The result of this phase is then used in the *apply* phase to update u's state. Finally, the *scatter* phase uses u's new state to update its adjacent edges.

With a graph model like GAS, a graph algorithm can be succinctly expressed in three functions, without having to worry about managing data layout and partitioning, or about scheduling parallel executions on multiple cores and multiple machines. A graph engine can then judiciously optimize data layout for efficient graph data access, partition the data in a way that reduces cross-core or cross-server communication, and achieve balanced parallelism for scaling and efficiency. For example, PowerGraph introduces vertex-cut to achieve balanced partitioning of graph data, resulting in improved scalability even for power-law graphs. In our experience, these optimizations are effective for machine learning algorithms; further, they need only be implemented once per engine rather than redundantly for each algorithm.

Machine learning on graphs. Machine learning is widely used in web search, recommendation systems, document analysis, and computational advertising. These algorithms learn models by training on data samples consisting of features. The goal of machine learning can often be expressed via an *objective function* with *parameters* that represent a model. This objective function captures the properties of the learned model, such as the error it incurs when predicting the probability that a user will click on an advertisement given that user's search query. The learning algorithm typically minimizes the objective function to obtain the model. It starts from an initial model and then iteratively refines the model by processing the training data, possibly multiple times.

Many machine learning problems can be modeled naturally and efficiently with graphs and solved by iterative convergence algorithms. For example, the Matrix Factorization (MF) algorithm [16], often used in recommendation systems, can be modeled as a computation on a bipartite user-item graph where each vertex corresponds to a user or an item and each edge corresponds to a user's rating of an item. As another example, a topic-modeling algorithm like LDA performs operations on a document-word graph where documents and words are vertices. If a document contains a word, there is an edge between them; the data on that edge are the topics of the word in the document. For many machine learning algorithms described as computations on a sparse matrix, the computation can often be easily transformed to operations on a graph representation of the sparse matrix. For example,



(b) Logistic Regression

Figure 1: Examples of machine learning on graphs.

in Logistic Regression (LR) the parameters of the model are maintained in a weight vector with each element being the weight of the corresponding feature. Each training sample is a sparse feature vector with each element being the value of a specific feature. The entire set of training samples can be treated as a sparse matrix with one dimension being the samples and the other being the features. If a sample i contains a value for feature j, the element (i, j) of the matrix is the value. Therefore, the data can also be modeled as a graph with samples and features being vertices. Weights are the data associated with feature vertices, and the feature values in each training sample are the data on edges. Figure 1 illustrates how MF and LR are modeled by graphs.

Gaps. Even though these machine learning algorithms can be cast in graph models, we observe gaps in current graph engines that preclude supporting them naturally and efficiently. These gaps involve data models, programming models, and execution scheduling.

Data models: The standard graph model assumes a homogeneous set of vertices, but the graphs that model machine learning problems often naturally have different types of vertices playing distinct roles (e.g., user vertices and item vertices). A heterogeneity-aware data model and layout is critical to performance.

Programming models: For machine learning computations, an iteration of a graph computation might involve multiple rounds of propagations between different types of vertices, rather than a simple series of GAS phases. The standard GAS model is unable to express such computation patterns efficiently. This is the case for LR, where the data (weights) of the feature vertices are first propagated to sample vertices to compute the objective function, with the gradients propagated back to feature vertices to update the weights. Implementing this process in GAS would unnecessarily require two consecutive GAS phases, with two barriers.

Execution scheduling: Machine learning frameworks have been shown to benefit from the Stale Synchronous Parallel (SSP) model, a relaxed consistency model with bounded staleness to improve parallelism. This is because machine learning algorithms typically describe the process to converge to a "good" solution according to an objective function and the convergence process itself is robust to variations and slack that can be leveraged to improve efficiency and parallelism. The mini-batch is another important scheduling concept, often used in stochastic gradient descent (SGD), where a small batch of samples are processed together to improve efficiency at the expense of slower convergence with respect to the number of iterations. Mini-batch size is an important parameter for those algorithms and needs to be tuned to find the best balance. Graph engines typically operate on individual vertices [29], or define an "iteration" or a batch on the entire graph [30], while mini-batches offer the additional flexibility to be in between.

TuX^2 therefore supports and optimizes for heterogeneity in the data model, advocates a new graph model that allows flexible composition of stages, and supports SSP and mini-batches in execution scheduling.

3 TuX^2 Design

TuX^2 is designed to preserve the benefits of graph engines while extending their data models, programming models, and scheduling approaches in service to distributed machine learning.

TuX^2 uses the *vertex-cut* approach, in which the edge set of a (high-degree) vertex can be split into multiple partitions, each maintaining a replica of the vertex. One of these replicas is designated the *master*; it maintains the master version of the vertex's data. All remaining replicas are called *mirrors*, and each maintains a local cached copy. We adopt vertex-cut because it is proven effective in handling power-law graphs and it connects naturally to the parameter-server model [26, 11]: The master versions of all vertices' data can be treated as the (distributed) global state stored in a parameter server. In each partition, TuX^2 maintains vertices and edges in separate arrays. Edges in the edge array are grouped by source vertex. Each vertex has an index giving the offset of its edge-set in the edge array. Each edge contains information such as the id of the partition containing the destination vertex and the index of that vertex in the corresponding vertex array. This graph data structure is optimized for traversal and outperforms vertex indexing using a lookup table.

Each partition is managed by a process that logically plays both a worker role, to enumerate vertices in the partition and propagate vertex data along edges, and a server role, to synchronize states between m and their corresponding masters. Inside a pr uses multiple threads for parallelization and the server and worker roles of a partition thread. Each thread is then responsible f ing a subset of mirror vertices for local com maintaining the states of a subset of master v

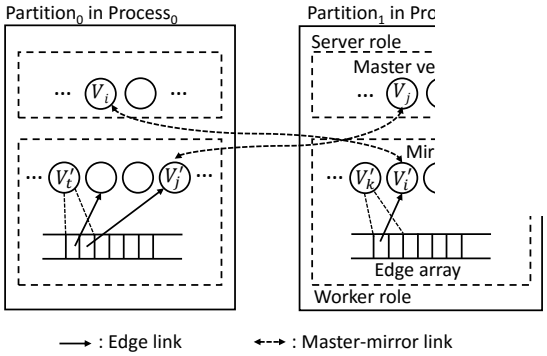


Figure 2: Graph placement and execution roles in TuX^2

3.1 Heterogeneous Data Layout

While traditional graph engines simply assume a homogeneous graph, TuX^2 supports heterogeneity in multiple dimensions of data layout, including vertex type and partitioning approach; it even supports heterogeneity between master and mirror vertex data types. Support for heterogeneity translates into significant performance gains (40%) in our evaluation (§5.2).

We highlight optimizations on bipartite graphs because many machine learning problems map naturally to bipartite graphs with two disjoint sets of vertices, e.g., users and items in MF, features and samples in LR, and so on. The two sets of vertices therefore often have different properties. For example, in the case of LR, only feature vertices contain a weight field and only sample vertices contain a target label field. And, in variants of LR like BlockPG [27], feature vertices also maintain extra history information. TuX^2 therefore allows users to define different vertex types, and places different types of vertex in separate arrays. This leads to compact data representation, thereby improving data locality during computation. Furthermore, different vertex types may have vastly different degrees. For example, in a user-item graph, item vertices can have links to thousands of users but user vertices typically only link to tens of items. TuX^2 uses bipartite-graph aware partitioning algorithms proposed in PowerLyra [7] and BiGraph [8] so that only high-degree vertices have mirror versions.

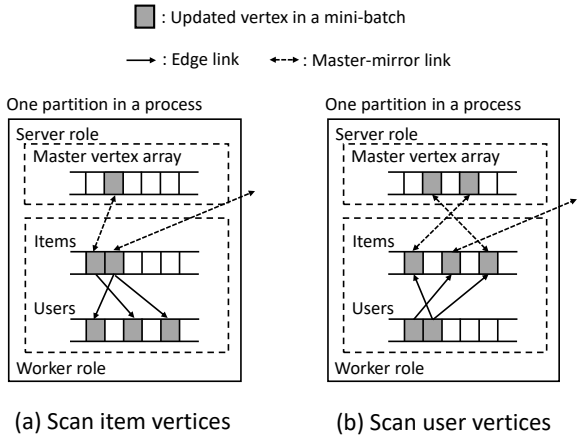


Figure 3: Example showing how separate vertex arrays are used for an MF bipartite graph. Edge arrays are omitted for conciseness.

In a bipartite graph, TuX^2 can enumerate all edges by scanning only vertices of one type. The choice of which type to enumerate sometimes has significant performance implications. Scanning the vertices with mirrors in a mini-batch tends to lead to a more efficient synchronization step as long as TuX^2 can identify the set of mirrors that have updates to synchronize with their masters, because these vertices are placed contiguously in an array. In contrast, if TuX^2 scans vertices without mirrors in a mini-batch, the mirrors that get updated for the other vertex type during the scan will be scattered and thus more expensive to locate. TuX^2 therefore allows users to specify which set of vertices to enumerate during the computation.

Figure 3 illustrates how TuX^2 organizes vertex data for a bipartite graph, using MF on a user-item graph as an example. Because user vertices have much smaller degree in general, only item vertices are split by vertex-cut partitioning. Therefore, a master vertex array in the server role contains only item vertices, and the worker role only manages user vertices. This way, there are no mirror replicas of user vertices and no distributed synchronization is needed. In the worker role, the mirrors of item and user vertices are stored in two separate arrays.

The figure also shows the benefit of scanning item vertices in a mini-batch. As shown in Figure 3(a), this leads to updated mirror vertices being located contiguously in an item vertex array. TuX^2 can therefore easily identify them for master-mirror synchronization by simply rescanning the corresponding range of that array. In contrast, scanning user vertices in a mini-batch would require an extra index structure to identify the mirror up-

dates. This is because they are scattered in an item vertex array as shown in Figure 3(b). Such an index structure would introduce extra overhead.

Another type of heterogeneity comes from different computations performed on master and mirror replicas of vertices, which may require different data structures for synchronization efficiency. For example, the BlockPG algorithm accesses and updates weights of a block of features in a mini-batch, while the objective function computed at sample vertices might depend on weights of features not in this block. This leads to auxiliary feature vertex attributes on mirrors, to record the historical deltas of feature weights to compute the value of the objective function incrementally. However, this delta attribute is not needed on masters, and hence does not need to be exchanged during synchronization. Similarly, a master vertex also maintains some extra attributes that are not needed on mirrors. TuX^2 therefore allows users to define different data structures for the master and mirror replicas of the same vertex.

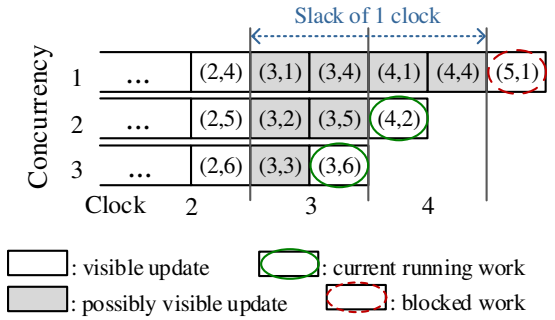


Figure 4: SSP with bounded staleness. A block labeled (i, j) indicates a task with id j in clock i.

3.2 Scheduling with SSP

TuX^2 supports the Stale Synchronous Parallel (SSP) model [11] with bounded staleness and mini-batches. SSP is based on the notion of *work-per-clock*, where a clock corresponds to an iteration over a mini-batch executed by a set of concurrent tasks. Iterative batch processing can be considered as a special case in which each iteration uses all input data. SSP introduces an explicit *slack* parameter, which specifies in clocks how stale a task's view of the globally shared state can be. The slack thus dictates how far ahead of the slowest task any task may progress. With a slack of s, a task at clock t is guaranteed to see all updates from clocks 1 to $t-s-1$, and it may see the updates from clocks $t-s$ to $t-1$. Figure 4 illustrates an SSP execution with a slack of 1.

TuX^2 executes each iteration on a mini-batch with a specified size. Each worker first chooses a set of vertices or edges as the current mini-batch to execute on. After the execution on the mini-batch finishes, TuX^2 acquires another set of vertices or edges for the next mini-batch, often by continuing to enumerate contiguous segments of vertex or edge arrays. TuX^2 supports SSP in the mini-batch granularity. It tracks the progress of each mini-batch iteration to enable computation of clocks. A worker considers clock t completed if the corresponding mini-batch is completed on all workers (including synchronizations between masters and mirrors) and if the resulting update has been applied to and reflected in the state. A worker can execute a task at clock t only if it knows that all clocks up to $t-s-1$ have completed, where s is the allowed slack.

3.3 MEGA Model in TuX^2

TuX^2 introduces a new stage-based *MEGA* model, where each stage is a computation on a set of vertices and their edges in a graph. Each stage has user-defined functions (UDF) to be applied on the vertices or edges accessed during it. TuX^2 supports four types of stage: Mini-batch, Exchange, GlobalSync, and Apply (hence the name MEGA); it allows users to construct an arbitrary sequence of stages. The engine is responsible for scheduling parallel executions of the UDFs on multiple cores and/or machines in each stage.

The MEGA model preserves the simplicity of the GAS model, while introducing additional flexibility to address deficiencies of the GAS model in supporting machine learning algorithms. For example, in algorithms such as MF and LDA, processing an edge involves updating both vertices. This requires two GAS phases, but can be accomplished in one Exchange phase in our model. For LR, the vertex data propagations in both directions should be followed by an *Apply* phase, but no *Scatter* phases are necessary; this can be avoided in the MEGA model because MEGA allows an arbitrary sequence of stages. We elaborate on the different types of stages next.

Exchange: This stage enumerates edges of a set of vertices, taking a UDF with the following signature:

$$Exchange(D_u, a_u, D_{(u,v)}, a_{(u,v)}, D_v, a_v, \tau)$$

`Exchange()` is performed on each enumerated edge. D_u and D_v are the data on vertices u and v, respectively. $D_{(u,v)}$ is the data associated with the edge (u,v). a_u, a_v, and $a_{(u,v)}$ are the corresponding accumulated deltas of the vertex and edge data, and τ is a user-defined shared context associated with each worker thread and maintained during the execution of the entire computation. All these parameters are allowed to be updated in this UDF. Users can use it to generate new accumulated deltas for vertices and edges, or to update their states directly. Given the vertex-cut graph placement, `Exchange()` may only update the mirror version data

(i.e., the local states) of the vertices. Users can also use τ to compute and store some algorithm-specific non-graph context data, which may be shared through global aggregation. By default, vertices not specified for enumeration are protected by vertex-level locks, but TuX^2 also allows users to implement their own lock-free semantics for some applications [14, 21, 37]. This stage is more flexible than the Gather/Scatter phases in the GAS model in that it does not imply or enforce a direction of vertex data propagation along an edge, and it can update the states of both vertices in the same UDF. It thereby improves efficiency for algorithms such as LDA and MF.

Apply: This stage enumerates a set of vertices and synchronizes their master and mirror versions. For each vertex, the master accumulates deltas from the mirrors, invokes $Apply(D_u, a_u, \tau)$ to update its global state, then updates the states on the mirrors. To support heterogeneity between master and mirror, TuX^2 allows users to define a base class `VertexDataSync` for the global state of a vertex that needs to be synchronized; masters and mirrors can define different subclasses, each inheriting from the base class, to include other information. The engine synchronizes only the data in `VertexDataSync` between master and mirror vertices.

GlobalSync: This stage is responsible for synchronizing the contexts τ across worker threads and/or aggregating the data across a set of vertices. There are three UDFs associated with this stage:

$$\tau^{i+1} \leftarrow Aggregate(D_v, \tau^i)$$

$$\tau^l \leftarrow Combine(\tau^i, \tau^j)$$

$$\tau^{i+1} \leftarrow Apply(\tau^i)$$

`Aggregate()` aggregates data across vertices into worker context τ. `Combine()` aggregates context τ across workers into a special worker, which maintains multiple versions of context τ for different clocks to support SSP. `Apply()` finalizes the globally aggregated τ (e.g., for re-scaling). After the execution of `Apply()`, the final aggregated τ is synchronized back to all workers. If the `Aggregate()` function is not provided, this stage will aggregate and synchronize the contexts τ only across workers.

Mini-Batch: This is a composite stage containing a sequence of other stages; it defines the stages to be executed iteratively for each mini-batch. `MiniBatch` defines the mini-batch size in terms of the number of vertices or edges to enumerate in each mini-batch, and, in the case of bipartite graphs, which type of vertex to enumerate (see examples in §4).

```
void StageSequenceBuilder(ExecStages){
    ExecStages.Add(ExchangeStage);
    ExecStages.Add(ApplyStage);
    ExecStages.Add(GlobalSyncStage);
}
```

(a) MF stage sequence for a batch

```
void StageSequenceBuilder(ExecStages){
    val mbStage = new MiniBatchStage;
    mbStage.SetBatchSize(1000, asEdge);
    mbStage.Add(ExchangeStage);
    mbStage.Add(ApplyStage);

    ExecStages.Add(mbStage);
    ExecStages.Add(GlobalSyncStage);
}
```

(b) MF stage sequence for a mini-batch

```
//ExchangeStage::
Exchange(v_user, v_item, edge,
         a_user, a_item, context){
    val pred = PredictRating(v_user,v_item);
    val loss = pred - edge.rating;
    context.loss += loss^2;
    (a_user,a_item) +=
        Gradient(loss,v_user,v_item);
}

//ApplyStage::
Apply(ver, accum, ctx){
    //Apply accumulated gradient
    ver.data += accum;
}

//GlobalSyncStage::
Combine(ctx1, ctx2){
    ctx.loss = ctx1.loss + ctx2.loss;
    return ctx;
}
```

(c) MF UDFs for each stage

Figure 5: Programming MF with the MEGA model

4 ML Algorithms on TuX^2

In this section, we detail how three machine learning algorithms are expressed and implemented in TuX^2.

Matrix Factorization (MF). MF, commonly used in recommendation systems, aims to decompose an adjacency matrix $M_{|U|\times|I|}$, where U is the set of users, I is the set of items, and the entry (u,i) is user u's rating on item i, into two matrices L and R, making M approximately equal to $L \times R$. TuX^2 models training data as a bipartite graph with users and items being vertices and user-item ratings being edges, and solves MF using SGD [16].

Figure 5 illustrates how MF is implemented in the MEGA model. `StageSequenceBuilder()` builds the stage sequence for each MF iteration. For MF in batch mode (Figure 5a), an iteration is composed of `ExchangeStage`, `ApplyStage`, and `GlobalSyncStage`. `Exchange()` (Figure 5c) computes the gradients of the loss function given a user and an item, and accumulates the gradients into `a_user` and `a_item`, respectively. In `Apply()`, the accumu-

```
//ExchangeStage::
Exchange(v_doc, v_word, edge,
         a_doc, a_word, context){
    val old_topic = edge.topic
    val new_topic = GibbsSampling(context,
                        v_doc, v_word)
    edge.topic = new_topic;

    //topic accumulator
    a_doc[old_topic]--;
    a_doc[new_topic]++;
    a_word[old_topic]--;
    a_word[new_topic]++;
    //update topic summary
    context.topic_sum[old_topic]--;
    context.topic_sum[new_topic]++;
}

//ApplyStage::
Apply(ver, accum, ctx){
    //Apply accumulated topic changes
    ver.topics += accum;
}

//GlobalSyncStage::
Combine(ctx1, ctx2){
    ctx.topic_sum
        = ctx1.topic_sum + ctx2.topic_sum;
    return ctx;
}
```

Figure 6: Programming LDA with the MEGA model

lated gradient is used to update the data of a vertex (a user or an item). `Combine()` sums the losses to evaluate convergence. For the mini-batch version (Figure 5b), only `ExchangeStage` and `ApplyStage` are performed per mini-batch, while `GlobalSyncStage` is conducted per iteration. The mini-batch size is set as the number of edges because each edge with its connected user and item vertices forms a training sample.

Latent Dirichlet Allocation (LDA). When applied to topic modeling, LDA trains on a set of documents to learn document-topic and word-topic distributions and thereby learn how to deduce any document's topics. TuX^2 implements SparseLDA [45], a widely used algorithm for large-scale distributed LDA training. In our graph model, vertices represent documents and words, while each edge between a document and a word means the document contains the word.

LDA's stage sequence is the same as MF's. Figure 6 shows the UDFs of the stages. Each edge is initialized with a randomly assigned topic. Each vertex (document or word) maintains a vector to track its distribution of all topics. The topic distribution of a vertex is the topic summary of the edges it connects. We also have a global topic summary maintained in a shared context. The computation iterates over the graph following the stage sequence until convergence. `Exchange()` performs Gibbs sampling [19] on each edge to compute a new topic for the edge. The new edge topic also changes the topic distributions of the vertices, as well as the topic

```
void StageSequenceBuilder(ExecStages){
    val mbStage = new MiniBatchStage;
    mbStage.SetBatchSize(1000,asVertex,
                         "feature");
    mbStage.Add(ExchangeStage0);
    mbStage.Add(ApplyStage);
    mbStage.Add(ExchangeStage1);

    ExecStages.Add(mbStage);
    ExecStages.Add(GlobalSyncStage);
}
```

(a) BlockPG stage sequence

```
//ExchangeStage0::
Exchange0(v_feature, v_sample, edge,
          a_feature, a_sample, ctx){
    (a_feature.g, a_feature.u) +=
        FeatureGradient(v_feature, v_sample)
}

//ExchangeStage1::
Exchange1(v_feature, v_sample, edge,
          a_feature, a_sample, ctx){
    v_sample.dual *=
        SampleDual(v_feature, v_sample)
}

//ApplyStage::
Apply(v_feature, a_feature, ctx){
    v_feature.weight +=
        SolveProximal(v_feature,a_feature,ctx);
}

//GlobalSyncStage::
Aggregate(ver, ctx){
    ctx.obj += CalcObj(ver);
}
Combine(ctx1, ctx2){
    ctx.obj = ctx1.obj + ctx2.obj;
    return ctx;
}
```

(b) BlockPG UDFs for stages

Figure 7: Programming BlockPG with the MEGA model

summary in the shared context; these changes are accumulated. `Apply()` applies the aggregated topic changes for each vertex. `Combine()` synchronizes the global topic summary among all workers.

Block Proximal Gradient (BlockPG). BlockPG [26, 27, 28] is a state-of-the-art logistic regression algorithm. It is modeled as a bipartite graph with features and samples as vertices and an edge between a feature and a sample vertex indicating that the sample contains the feature.

Figure 7b shows the pseudocode of BlockPG's UDFs. BlockPG randomly divides features into blocks and enumerates each block as a mini-batch. Each mini-batch involves two `Exchange*()` stages with an `Apply()` stage in between. `Exchange0()` calculates and accumulates for each edge both the gradient and the diagonal part of the second derivative for the corresponding feature vertex. `Apply()` then synchronizes the vertices' accumulated values into the master feature vertices to update their weights using a proximal operator [34]. Then, `Exchange1()` uses the new weights of features

to compute new states of samples. Note that BlockPG does not need `Apply()` for sample vertices. TuX^2 therefore optimizes `Apply()` by setting the mini-batch size in terms of the number of feature vertices (as shown in Figure 7a) and partitioning the graph to not cut the sample vertices. Also, only the weight value needs to be synchronized between master and mirror feature vertices. TuX^2 allows a master vertex to maintain private data that does not get synchronized to mirrors, e.g., information for the proximal operator.

5 Implementation and Evaluation

We implemented TuX^2 in about 12,000 lines of C++ code. It can be built and deployed on both Linux and Windows clusters with each machine running one TuX^2 process. The system is entirely symmetric: All the processes participating in the computation are peers executing the same binary. TuX^2 takes graph data in a collection of text files as input. Each process picks a separate subset of those files and performs bipartite-graph-aware algorithms [7, 8] to partition the graph in a distributed way. Each partition is assigned to, and stored locally with, a process. The data in each partition are placed as they are loaded and used in computation. For interprocess communication, TuX^2 uses a network library that supports both RDMA and TCP.

In the rest of this section, we present detailed evaluation results to support our design choices and to demonstrate the benefits of supporting machine learning on graph engines. We compare TuX^2 with state-of-the-art graph systems PowerGraph [17, 4] and PowerLyra [7] and ML systems Petuum [20, 3] and Parameter Server [26, 2].

Experimental setup. We conduct most of our experiments on a commodity cluster with 32 servers. Each server is equipped with dual 2.6 GHz Intel Xeon E5-2650 processors (16 physical cores), 256 GB of memory, and a Mellanox ConnectX-3 InfiniBand NIC with 54 Gbps bandwidth. TuX^2 uses RDMA by default, but uses TCP when comparing to other systems for fairness.

To evaluate TuX^2, we have fully implemented the MF, LDA, and BlockPG algorithms introduced in §4, setting the feature dimension of MF to 50 and the topic count of LDA to 100 in all experiments. The algorithms are selected to be representative and cover a spectrum, ranging from computation-bound (e.g., LDA) to communication-bound (e.g., BlockPG). Table 1 lists the datasets that we use for evaluation. NewsData and AdsData are two real datasets used in production by Microsoft for news and advertisement. Netflix [6] is the largest public dataset that is available for MF. We also generate a larger synthesized dataset, which is the default dataset for MF experiments. All performance numbers in our experiments are calculated by averaging over 100 iterations; in all cases we observed very little variation.

Dataset name	# of users/ docs/samples	# of items/ words/features	# of edges
NewsData (LDA)	7.3 M	418.4 K	1.4 B
AdsData (BlockPG)	924.8 M	209.3 M	64.9 B
Netflix (MF)	480.2 K	17.8 K	100.5 M
Synthesized (MF)	30 M	1 M	6.3 B

Table 1: Datasets (K: thousand, M: million, B: billion).

Algorithm	ML systems	TuX^2	LOC reduction
MF (Petuum)	> 300	50	83%
LDA (Petuum)	> 950	252	73%
BlockPG (PS)	> 350	79	77%

Table 2: Algorithm implementations in lines of code, ML systems vs. TuX^2. (PS: Parameter Server)

5.1 Programmability

By providing a high-level MEGA graph model, TuX^2 makes it significantly easier to write distributed machine learning algorithms, relieving developers from handling the details of data organization, enumeration, partitioning, parallelism, and thread management. As one indication, Table 2 shows the significant reduction (73–83%) in lines of code (LOC) to implement the three algorithms in TuX^2, compared to the C++ implementations of the same algorithms on Petuum or Parameter Server. The lines of code are comparable to those written in the GAS model.

5.2 Managing ML Data as a Graph

Data layout. Data layout matters greatly in the performance of machine learning algorithms. Figure 8 compares the performance of BlockPG, MF, and LDA with two different layouts: one an array-based graph data layout in TuX^2 and the other a hash-table-based lay-

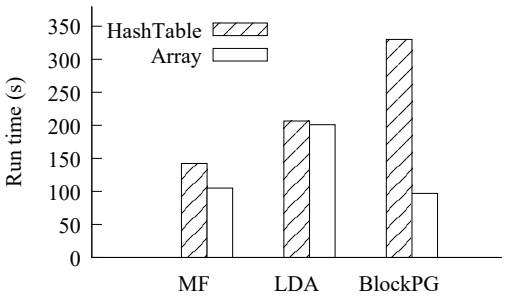


Figure 8: Effect of data layout (32 servers)

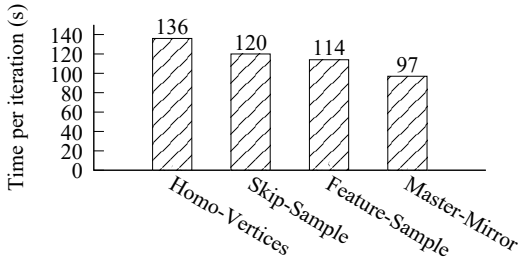


Figure 9: Effect of heterogeneity (BlockPG, 32 servers)

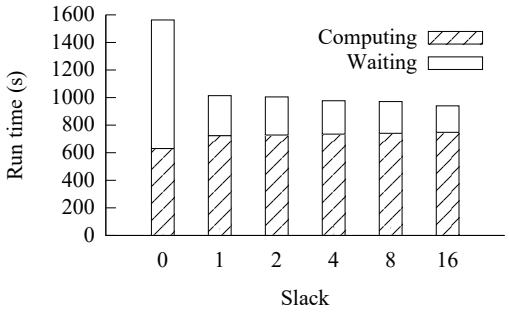


Figure 10: Run time, with breakdown, to converge to the same point under different slack (MF, 32 servers)

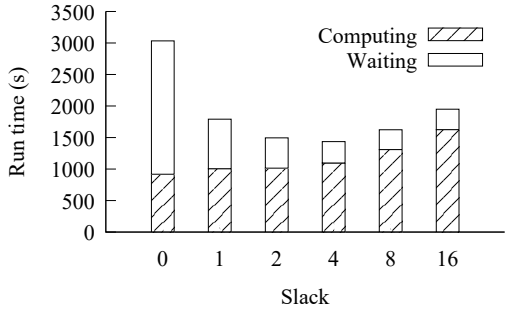


Figure 11: Run time, with breakdown ,to converge to the same point under different slack (BlockPG, 32 servers)

out often used in parameter-server-based systems (but implemented in TuX^2 for comparison). The y-axis is the average running time of one iteration for BlockPG, and of 10 iterations for MF and LDA to show the numbers on a similar scale. These results show that the graph layout improves performance by up to 2.4× over the hash-table-based layout. We observe smaller improvement in LDA because LDA involves more CPU-intensive floating-point computation, making data access contribute a smaller portion of overall run time.

Heterogeneity. Supporting heterogeneity in TuX^2 is critical to the performance of machine learning algorithms. We evaluate the benefits of supporting different dimensions of vertex heterogeneity using BlockPG on 32 servers. As shown in Figure 9, if we model all vertices as the same vertex type, each iteration takes 136 s (Homo-Vertices in the figure). For BlockPG, because only feature vertices have mirrors, we can specify to enumerate only feature vertices in each mini-batch and not track whether sample vertices are updated because they do not have mirrors to synchronize (see §3.1). This setup leads to a reduction of 16 s per iteration (Skip-Sample in the figure). Next, if we define different vertex data types for features and samples for a more compact representation, each iteration can save an additional 6 s (Feature-Sample in the figure). Finally, as discussed in §3.3, we can allow masters and mirrors to have different types and can indicate which data need to be synchronized. Doing this makes each iteration take only 97 s (Master-Mirror in the figure), a total performance improvement of 40% over the original homogeneous setting.

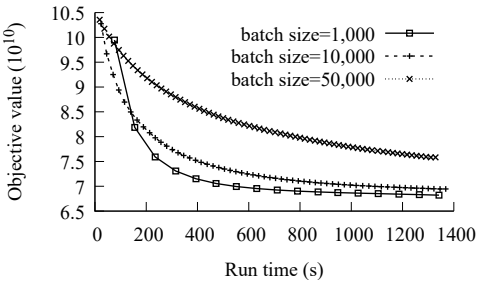
5.3 Extensions for Machine Learning

SSP slack and mini-batch size can be configured in TuX^2 to tune algorithm convergence.

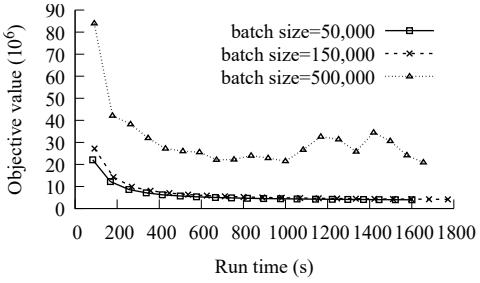
Stale Synchronous Parallel. TuX^2 supports the configuration of slack as a staleness bound for SSP, to allow users to tune the parameter for desirable convergence. The effect of slack varies by algorithm. For MF, as shown in Figure 10, the overall convergence accelerates as the slack increases. The breakdown confirms that increasing slack reduces waiting time, while increasing computing time only slightly, indicating that it takes about the same (or a slightly larger) number of iterations to reach the same convergence point. For BlockPG, however, as shown in Figure 11, computing time increases significantly as slack increases to 8 and 16, indicating that it is taking many more iterations for BlockPG to converge when slack is larger. A slack value of 4 is the optimal point in terms of overall execution time.

Mini-Batch. Mini-batch size is another important parameter that TuX^2 lets users tune, since it also affects convergence, as we show in this experiment for MF and BlockPG. (LDA is inherently a batch algorithm and does not support mini-batches.) Figure 12 shows the convergence (to objective values) over time with slack set to 16 on 32 servers. We show each iteration as a point on the corresponding curve to demonstrate the effect of mini-batch size on the execution time of each iteration.

For MF, as shown in Figure 12a, we see that convergence with a smaller mini-batch size (e.g., 1,000) is much faster than that with a larger one (e.g., 10,000). However, a smaller mini-batch size could introduce more frequent communication, slowing down the computation in each iteration significantly, as confirmed by more



(a) MF, 32 servers



(b) BlockPG, 32 servers

Figure 12: Convergence with varying mini-batch size

sparse iteration points on the curve for mini-batch size 1,000. This is why we also observe that convergence with mini-batch size 1,000 is worse than that with 10,000 during the first 180 s. Similar results can be observed for BlockPG in Figure 12b. For BlockPG, an improper batch size could even make it non-convergent, as is the case when batch size is 500,000.

5.4 System Performance

TuX^2 vs. PowerGraph and PowerLyra. We first compare TuX^2 with PowerGraph and its successor PowerLyra, which support a GAS-based MF implementation. Because PowerGraph and PowerLyra do not support SSP or mini-batch, for fairness we configure TuX^2 to use a batched MF implementation with no slack. We run MF on the Netflix dataset, and Figure 13 shows the performance comparison of TuX^2, PowerGraph, and PowerLyra with different numbers of servers (each with 16 threads).

The figure shows that, consistent with the results reported in PowerLyra [7], PowerLyra outperforms PowerGraph in the multi-server cases (by 1.6x) due to a better partitioning algorithm that leads to a lower vertex replication factor. TuX^2 outperforms both PowerGraph and PowerLyra by more than an order of magnitude. The huge performance gap is largely due to our flexible MEGA model. Specifically, in PowerGraph and PowerLyra, the computation per iteration for MF is composed of two GAS phases, one for updating the user vertices and the other for the item vertices. This introduces significant synchronization overhead and some unnecessary stages due to the constraints of the GAS model. In contrast, TuX^2 needs only an `ExchangeStage` and an `ApplyStage` for each iteration in the MEGA model. Our detailed profiling on one iteration in the 8-server experiment further shows that, while the Exchange phase (which calculates the gradients) in TuX^2 takes only 0.5 s, the corresponding Gather phase takes 1.6 s in the GAS model. The difference is mainly due to TuX^2's heterogeneous data layout. Furthermore, the extra phases (i.e., the two Scatter phases) needed in the GAS model take an additional 7.4 s.

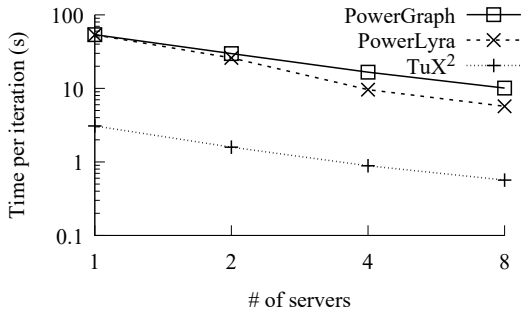
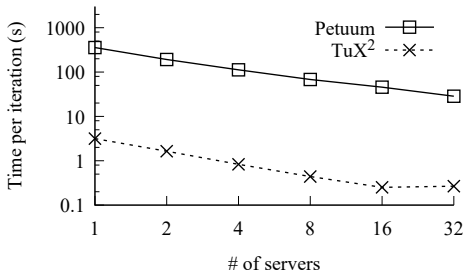


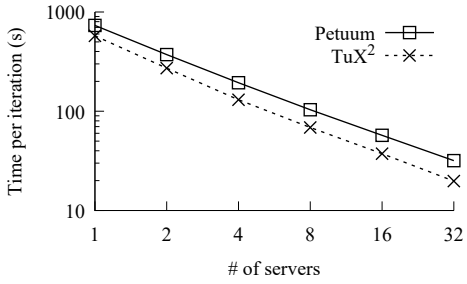
Figure 13: TuX^2 vs. PowerGraph/PowerLyra (MF, Netflix, log scale)

TuX^2 vs. machine learning systems. We compare TuX^2 with two state-of-the-art distributed machine learning systems: Petuum and Parameter Server (PS). We compare with Petuum using MF and LDA and we compare with PS using BlockPG. We have validated the results of our experiments to confirm that the algorithms in TuX^2 are the same as those in Petuum and in PS, respectively. We set slack to 0 as it produces a deterministic result every iteration, leading to the same convergence curve. We use time per iteration as our metric for comparison because the convergence per iteration is the same in this configuration. We evaluate on other configurations (not shown due to space constraints) and the results are similar. Compared with these systems, TuX^2, as a graph engine, inherits a series of graph-related optimizations for machine learning, such as efficient graph layout and balanced parallelism from vertex-cut partitioning. The following experiments evaluate these benefits of TuX^2.

Petuum: We compare Petuum and TuX^2 using MF and LDA because these two algorithms have been implemented in both Petuum and TuX^2. All the experiments are conducted on 32 servers with 16 threads per server. Figures 14a and 14b show the average execution time per

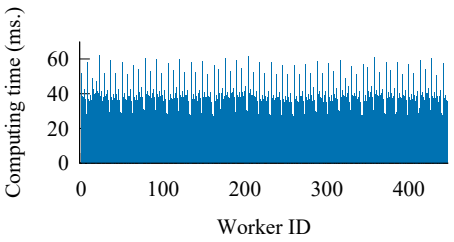


(a) MF, Netflix

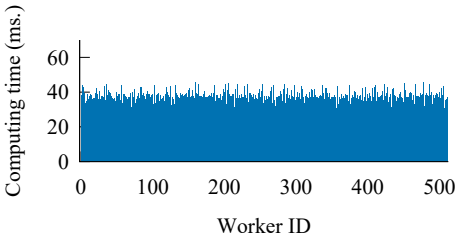


(b) LDA, NewsData

Figure 14: TuX^2 vs. Petuum (log scale)



(a) Imbalance in ParameterServer, 32 servers



(b) Balance in TuX^2, 32 servers

Figure 15: Mini-batch time across workers (BlockPG)

iteration of MF and LDA in Petuum and TuX^2 with different numbers of servers.

For MF, TuX^2 outperforms Petuum by two orders of magnitude, due to two main reasons. First, Petuum's distributed shared memory table, implemented in a multi-layer hash structure, introduces significant overhead, even compared with our hash-table baseline used in §5.2. Petuum also does fine-grained row-level version tracking, causing a staleness check to be triggered for every read/write operation. In contrast, TuX^2 uses a worker-level staleness check when an iteration/mini-batch starts, as described in §3.2. Second, in Petuum, both user data and item data contain model parameters and are stored in the parameter server. Updating either type involves communication with the parameter server. It is worth pointing out that this is not a fundamental problem with Petuum's design and can be fixed by moving user data off the parameter server. (Based on our communication with the Petuum authors, this issue has already been fixed in the new version [42], but the fixed version is not yet publicly available.) TuX^2 partitions the bipartite graph in such a way that only item vertices have mirrors, making the updates on user vertices efficient without unnecessary communication. This is a natural configuration in TuX^2 that users can enable effortlessly, easily avoiding the problem we observe in this version of Petuum. Note that TuX^2 does not scale well from 16 to 32 servers for MF. This is because Netflix data is small when divided among 32 servers $\times$ 16 threads (only around 3 MB per thread), so communication cost starts to dominate, limiting further scaling.

For LDA, the graph layout benefit is smaller compared to that for MF. Figure 14b shows that TuX^2 outperforms Petuum in LDA by 27% (1 server) to 61% (32 servers). This is consistent with the layout experiment in §5.2, which was also affected by the CPU-intensive nature of the floating-point computation in LDA.

Parameter Server (PS): We compare PS with TuX^2 using BlockPG, as it is implemented in both systems. We set mini-batch size to 300,000 for both. For PS, based on our experimentation on different thread-pool configurations, we find the best configuration uses 14 worker threads and 2 server threads per machine. We therefore use this configuration in our experiments. Due to the large data size (64B edges) involved, the experiment is performed only on 32 servers. When operating on the AdsData dataset, BlockPG takes 125 s on average per iteration on TuX^2, compared to 186 s on PS, which is 48% longer.

Unlike with Petuum, data layout is not the main reason that TuX^2 outperforms PS: PS carefully customizes its data structure for BlockPG, which is largely on par with TuX^2's general graph layout. Handling the imbalance caused by data skew (e.g., where some features exist in a large number of samples) makes the most difference in this case. Figure 15a shows the execution time of one representative mini-batch for all worker threads

in PS. A few threads are shown to work longer than the others, forcing those others to wait at synchronization points. In contrast, TuX^2 employs vertex-cut even for threads inside the same process, a built-in feature of the graph engine, to alleviate imbalance. Figure 15b shows that TuX^2 achieves balanced execution for all threads. While SSP slack could also help alleviate the effect of imbalance, it usually leads to slower convergence. Our vertex-cut optimization does not affect convergence and is strictly better.

6 Related Work

TuX^2 builds upon a large body of research on iterative graph computation and distributed machine learning systems. Pregel [30] proposes the vertex-program model, which has been adopted and extended in subsequent work, such as GraphLab [29] and PowerGraph [17]. TuX^2 uses the vertex-cut model proposed in PowerGraph and applies it also to partitioning within the process for balanced thread parallelism. It also incorporates bipartite-graph-specific partitioning schemes proposed in PowerLyra [7] and BiGraph [8] with further optimizations of computation. By connecting graph models to machine learning, our work makes advances in graph computation relevant to machine learning. This includes optimizations on graph layout, sequential data access, and secondary storage (e.g., GraphChi [24], Grace [36], XStream [39], Chaos [38], and FlashGraph [47]), distributed shared memory and RDMA (e.g., Grappa [32] and GraM [43]), and NUMA-awareness, scheduling, and load balancing (e.g., Galois [33], Mizan [22], and Polymer [46]).

TuX^2's design is influenced by parameter-server-based distributed machine learning, which was initially proposed and evolved to scale specific machine learning applications such as LDA [40, 5] and deep learning [15]. Petuum [13, 20, 42, 44] and Parameter Server [26] move towards general platforms, incorporate flexible consistency models, and improve scalability and efficiency. Petuum and its subsequent work on STRADS [23, 25] further propose to incorporate optimizations such as model parallelism, uneven convergence, and error tolerance. Many of these can be integrated into a graph engine like TuX^2, allowing users to benefit from both graph and machine learning optimizations, a future direction that we plan to explore further. We also see a trend where some of the design and benefits in graph systems have found their way into these machine learning systems (e.g., optimized layout in Parameter Server's BlockPG implementation and a high-level graph-model-like abstraction in STRADS), further supporting our theme of the convergence of the two. Parameter servers have also been proposed to support deep learning [9, 12, 15] and have been enhanced with GPU-specific optimizations in GeePS [12].

There is a large body of work on general distributed big-data computing platforms, including for example Mahout [1] on Hadoop and MLI [41] on Spark for machine learning on MapReduce-type frameworks. Piccolo [35] enables parallel in-memory computation on shared distributed, mutable state in a parameter-server-like interface. Another interesting research direction, pursued in GraphX [18] for example, explores how to support graph computation using a general data-flow engine. Naiad [31] introduces a new data-parallel dataflow model specifically for low-latency streaming and cyclic computations, which has also been shown to express graph computation and machine learning. Both have built known graph models, such as GAS, on top of their dataflow abstractions, while TuX^2 proposes a new graph model with important extensions for machine learning algorithms.

7 Conclusion

Through TuX^2, we advocate the convergence of graph computation and distributed machine learning. TuX^2 represents a critical step in this direction by showing not only the feasibility, but also the potential, of such convergence. We accomplish this by introducing important machine learning concepts to graph computation; defining a new, flexible graph model to express machine learning algorithms efficiently; and demonstrating the benefits through extensive evaluation on representative machine learning algorithms. Going forward, we hope that TuX^2 will provide a common foundation for further research in both graph computation and distributed machine learning, allowing more machine learning algorithms and optimizations to be expressed and implemented easily and efficiently at scale.

Acknowledgments

We thank our shepherd Adam Wierman and the anonymous reviewers for their valuable comments and suggestions. We are grateful to our colleague Jay Lorch, who carefully went through the paper and helped improve the quality of writing greatly. Jilong Xue was partially supported by NSF of China (No. 61472009).

References

[1] Apache foundation. mahout project. `http://mahout.apache.org`.

[2] ParameterServer. `https://github.com/dmlc/parameter_server`.

[3] Petuum v0.93. https://github.com/petuum/bosen/tree/release_0.93.

[4] PowerGraph v2.2. https://github.com/dato-code/PowerGraph.

[5] AHMED, A., ALY, M., GONZALEZ, J., NARAYANAMURTHY, S., AND SMOLA, A. J. Scalable inference in latent variable models. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining* (2012), WSDM'12, ACM.

[6] BENNETT, J., AND LANNING, S. The Netflix Prize. In *Proceedings of KDD cup and workshop* (2007), vol. 2007.

[7] CHEN, R., SHI, J., CHEN, Y., AND CHEN, H. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), EuroSys'15, ACM.

[8] CHEN, R., SHI, J., ZANG, B., AND GUAN, H. Bipartite-oriented distributed graph partitioning for big learning. In *Proceedings of 5th Asia-Pacific Workshop on Systems* (2014), APSys'14, ACM.

[9] CHILIMBI, T., SUZUE, Y., APACIBLE, J., AND KALYANARAMAN, K. Project Adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation* (2014), OSDI'14, USENIX.

[10] CHING, A., EDUNOV, S., KABILJO, M., LOGOTHETIS, D., AND MUTHUKRISHNAN, S. One trillion edges: Graph processing at Facebook-scale. *Proc. VLDB Endow. 8*, 12 (Aug. 2015).

[11] CUI, H., CIPAR, J., HO, Q., KIM, J. K., LEE, S., KUMAR, A., WEI, J., DAI, W., GANGER, G. R., GIBBONS, P. B., GIBSON, G. A., AND XING, E. P. Exploiting bounded staleness to speed up big data analytics. In *2014 USENIX Annual Technical Conference* (2014), USENIX ATC'14, USENIX.

[12] CUI, H., ZHANG, H., GANGER, G. R., GIBBONS, P. B., AND XING, E. P. GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), EuroSys'16, ACM.

[13] DAI, W., KUMAR, A., WEI, J., HO, Q., GIBSON, G., AND XING, E. P. High-performance distributed ML at scale through parameter server consistency models. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* (2015), AAAI'15, AAAI Press.

[14] DE SA, C. M., ZHANG, C., OLUKOTUN, K., RÉ, C., AND RÉ, C. Taming the wild: A unified analysis of hogwild-style algorithms. In *Advances in Neural Information Processing Systems 28* (2015), NIPS'15, Curran Associates, Inc.

[15] DEAN, J., CORRADO, G., MONGA, R., CHEN, K., DEVIN, M., MAO, M., AURELIO RANZATO, M., SENIOR, A., TUCKER, P., YANG, K., LE, Q. V., AND NG, A. Y. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25*, NIPS'12. Curran Associates, Inc., 2012.

[16] GEMULLA, R., NIJKAMP, E., HAAS, P. J., AND SISMANIS, Y. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2011), KDD'11, ACM.

[17] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation* (2012), OSDI'12, USENIX.

[18] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. GraphX: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation* (2014), OSDI'14, USENIX.

[19] GRIFFITHS, T. L., AND STEYVERS, M. Finding scientific topics. *Proceedings of the National Academy of Sciences 101*, suppl 1 (2004).

[20] HO, Q., CIPAR, J., CUI, H., LEE, S., KIM, J. K., GIBBONS, P. B., GIBSON, G. A., GANGER, G., AND XING, E. P. More effective distributed ML via a stale synchronous parallel parameter server. In *Advances in Neural Information Processing Systems 26*, NIPS'13. Curran Associates, Inc., 2013.

[21] JOHNSON, M., SAUNDERSON, J., AND WILLSKY, A. Analyzing Hogwild parallel Gaussian Gibbs sampling. In *Advances in Neural Information Processing Systems 26*, NIPS'13. Curran Associates, Inc., 2013.

[22] KHAYYAT, Z., AWARA, K., ALONAZI, A., JAMJOOM, H., WILLIAMS, D., AND KALNIS, P. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys'13, ACM.

[23] KIM, J. K., HO, Q., LEE, S., ZHENG, X., DAI, W., GIBSON, G. A., AND XING, E. P. STRADS: A distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), EuroSys'16, ACM.

[24] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. GraphChi: Large-scale graph computation on just a PC. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation* (2012), OSDI'12, USENIX.

[25] LEE, S., KIM, J. K., ZHENG, X., HO, Q., GIBSON, G. A., AND XING, E. P. On model parallelization and scheduling strategies for distributed machine learning. In *Advances in Neural Information Processing Systems 27* (2014), NIPS'14, Curran Associates, Inc.

[26] LI, M., ANDERSEN, D. G., PARK, J. W., SMOLA, A. J., AHMED, A., JOSIFOVSKI, V., LONG, J., SHEKITA, E. J., AND SU, B.-Y. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation* (2014), OSDI'14, USENIX.

[27] LI, M., ANDERSEN, D. G., AND SMOLA, A. J. Distributed delayed proximal gradient methods. In *NIPS Workshop on Optimization for Machine Learning* (2013).

[28] LI, M., ANDERSEN, D. G., SMOLA, A. J., AND YU, K. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems 27* (2014), NIPS'14, Curran Associates, Inc.

[29] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow. 5*, 8 (Apr. 2012).

[30] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (2010), SIGMOD'10, ACM.

[31] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP'13, ACM.

[32] NELSON, J., HOLT, B., MYERS, B., BRIGGS, P., CEZE, L., KAHAN, S., AND OSKIN, M. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference* (2015), USENIX ATC'15, USENIX.

[33] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP'13, ACM.

[34] PARIKH, N., AND BOYD, S. Proximal algorithms. *Found. Trends Optim. 1*, 3 (Jan. 2014).

[35] POWER, R., AND LI, J. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), OSDI'10, USENIX.

[36] PRABHAKARAN, V., WU, M., WENG, X., MCSHERRY, F., ZHOU, L., AND HARADASAN, M. Managing large graphs on multi-cores with graph awareness. In *2012 USENIX Annual Technical Conference* (2012), USENIX ATC'12, USENIX.

[37] RECHT, B., RE, C., WRIGHT, S., AND NIU, F. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems 24*, NIPS'11. Curran Associates, Inc., 2011.

[38] ROY, A., BINDSCHAEDLER, L., MALICEVIC, J., AND ZWAENEPOEL, W. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP'15, ACM.

[39] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP'13, ACM.

[40] SMOLA, A., AND NARAYANAMURTHY, S. An architecture for parallel topic models. *Proc. VLDB Endow. 3*, 1-2 (Sept. 2010).

[41] SPARKS, E. R., TALWALKAR, A., SMITH, V., KOTTALAM, J., PAN, X., GONZALEZ, J., FRANKLIN, M. J., JORDAN, M. I., AND KRASKA, T. MLI: An API for Distributed Machine Learning. In *2013 IEEE 13th International Conference on Data Mining* (2013), ICDM'13, IEEE.

[42] WEI, J., DAI, W., QIAO, A., HO, Q., CUI, H., GANGER, G. R., GIBBONS, P. B., GIBSON, G. A., AND XING, E. P. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (2015), SoCC'15, ACM.

[43] WU, M., YANG, F., XUE, J., XIAO, W., MIAO, Y., WEI, L., LIN, H., DAI, Y., AND ZHOU, L. GraM: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (2015), SoCC'15, ACM.

[44] XING, E. P., HO, Q., DAI, W., KIM, J.-K., WEI, J., LEE, S., ZHENG, X., XIE, P., KUMAR, A., AND YU, Y. Petuum: A new platform for distributed machine learning on big data. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2015), KDD'15, ACM.

[45] YAO, L., MIMNO, D., AND MCCALLUM, A. Efficient methods for topic model inference on streaming document collections. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2009), KDD'09, ACM.

[46] ZHANG, K., CHEN, R., AND CHEN, H. NUMA-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2015), PPoPP'15, ACM.

[47] ZHENG, D., MHEMBERE, D., BURNS, R., VOGELSTEIN, J., PRIEBE, C. E., AND SZALAY, A. S. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *13th USENIX Conference on File and Storage Technologies* (2015), FAST'15, USENIX.

Correct by Construction Networks using Stepwise Refinement

Leonid Ryzhyk* *VMware Research*
Nikolaj Bjørner *Microsoft Research*
Marco Canini *KAUST*
Jean-Baptiste Jeannin *Samsung Research America*
Cole Schlesinger* *Barefoot Networks*
Douglas B. Terry* *Amazon*
George Varghese *UCLA*

Abstract

Building software-defined network controllers is an exercise in software development and, as such, likely to introduce bugs. We present Cocoon, a framework for SDN development that facilitates both the design and verification of complex networks using *stepwise refinement* to move from a high-level specification to the final network implementation.

A Cocoon user specifies intermediate design levels in a hierarchical design process that delineates the modularity in complicated network forwarding and makes verification extremely efficient. For example, an enterprise network, equipped with VLANs, ACLs, and Level 2 and Level 3 Routing, can be decomposed cleanly into abstractions for each mechanism, and the resulting stepwise verification is over 200x faster than verifying the final implementation. Cocoon further separates static network design from its dynamically changing configuration. The former is verified at design time, while the latter is checked at run time using statically defined invariants. We present six different SDN use cases including B4 and F10. Our performance evaluation demonstrates that Cocoon is not only faster than existing verification tools but can also find many bugs statically before the network design has been fully specified.

1 Introduction

Software-defined networks (SDNs) are a popular and flexible means of implementing network control. In an SDN, a logically-centralized controller governs network behavior by emitting a stream of data-plane configurations in response to network events such as changing traffic patterns, new access control rules, intrusion detection, and so on. But decades of research and industry experience in software engineering have shown that writing bug-free software is far from trivial. By shifting to software, SDNs trade one form of complexity for another.

*Work performed at Samsung Research America.

Data-plane verification has risen in popularity with SDNs. As the controller generates new forwarding configurations, tools like Header Space Analysis (HSA) and Veriflow [16, 17] verify that safety properties hold for each configuration in real time. Network operators can rest assured that access control violations, routing loops, and other common misconfiguration errors will be detected before being deployed.

This style of verification is an important safeguard, but falls short in several ways.

Design. Applying verification techniques early in the development cycle saves effort by catching bugs as soon as they are introduced. But correctness properties often depend on many mechanisms spanning many different levels of abstraction and time scales. Thus the entire controller must be implemented before data-plane verification can be utilized. Furthermore, data-plane verification catches bugs once the controller has been deployed in a live network, making it hard to fix the bug without disrupting network operation.

Debugging. Verifying detailed, whole-network configurations makes debugging difficult: It is difficult to pinpoint which part of the controller caused a particular property violation in the final configuration [35].

Scalability. Although existing tools verify one property for a realistic network in under a second, the number of checks can scale non-linearly with network size. For example, checking connectivity between all pairs requires a quadratic number of verifier invocations [27]. Thus practical verification at scale remains elusive.

Ideally, the controller software itself might be statically verified to guarantee it never produces configurations that violate safety properties. But proving arbitrary software programs correct is a frontier problem. Recent work has proposed full controller verification, but only for controllers with limited functionality [3].

We propose a middle ground—a correct-by-construction SDN design framework that combines static verification with runtime checks to efficiently

verify complex SDNs, detecting most bugs at design time. Our framework, called Cocoon, for Correct by Construction Networking, consists of an SDN programming language, a verifier for this language, and a compiler from the language to data-plane languages: NetKAT [2] and P4 [4].

Cocoon is based on two principles. First, it enables SDN design by *stepwise refinement*. A network programmer begins by specifying a high-level view which captures the network's behavior from an end host perspective. Such a specification might say: "A packet is delivered to the destination host if and only if its sender is not blacklisted by the network security policy", while eliding details such as forwarding or access control mechanisms. In essence, this high-level view specifies correct network behavior. The network engineer continues by refining the underspecified parts of the design, filling in pieces until sufficient detail exists to deploy the network. A refined specification may state: "End hosts are connected via Ethernet switches to zone routers, which forward packets between zones via the core network, while dropping packets that violate security policy."

Cocoon automatically verifies that each refinement preserves the behavior of the higher-level view of the network by reducing each refinement to a Boogie program and using the Corral verifier to check this program for refinement violations [19]. Bugs are immediately detected and localized to the step in which they are introduced. The refinement relation is transitive, and so Cocoon guarantees that the lowest-level implementation refines the highest-level specification.

Second, Cocoon separates static network design from its run-time configuration. While refinements specify static invariants on network behavior, dynamic configuration is captured by *runtime-defined functions* (RDFs). In the above example the hosts and exact security policy are not known at design time and serve as design parameters. They are specified as RDFs, i.e., functions that are declared but not assigned a concrete definition at design time. RDFs are generated and updated at run time by multiple sources: the SDN controller reporting a new host joining, the network operator updating the security policy, an external load balancer redistributing traffic among redundant links, *etc.* Upon receiving an updated RDF definition, the Cocoon compiler generates a new data plane configuration.

To statically verify the design without knowing the exact configuration, Cocoon relies on static *assumptions*. At design time, RDFs can be annotated with assumptions that constrain their definitions. For example, the topology of the network may be updated as links come up and down, but each refinement may only need to know that the topology remains connected. At run time, Cocoon checks that RDF definitions meet their assumptions. This separation minimizes real-time verification cost: most of the effort has been done up-front at design time.

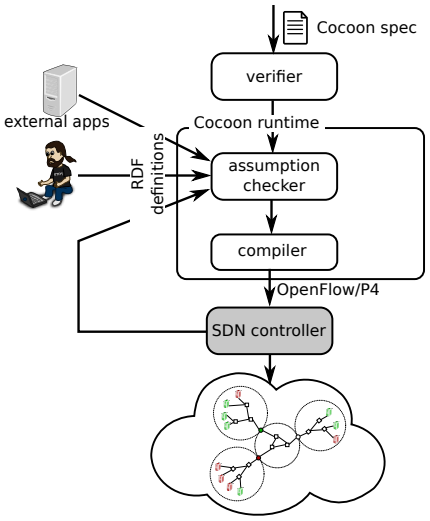


Figure 1: Cocoon architecture.

Hence, Cocoon decomposes verification into two parts, as shown in Figure 1. Static verification guarantees correctness of all refinements; this verification is done once, before network deployment. Dynamic verification checks that behaviors supplied at run time (by updating RDFs) meet the assumptions each refinement makes about run-time behaviors.

Contributions. The main contribution of this paper is a new network design and verification methodology based on stepwise refinement and separation of static and dynamic behavior, and the Cocoon language and runtime, which support this methodology. Cocoon is a language carefully designed to be both amenable to stepwise refinement-based verification and also able to capture a wide variety of networking behavior. Its design enables:

- Writing complex specifications easily by phrasing them as high-level network implementations.
- Faster verification of functional correctness, with stepwise refinement naturally helping to localize the source of errors.
- Natural composition with other verification tools, like HSA [16], NetPlumber [15] and Veriflow [17], improving the speed at which they can verify network properties.

We evaluate Cocoon by using it to design and verify six realistic network architectures. Our performance evaluation demonstrates that Cocoon is faster than existing data-plane verification tools, while also being able to find many defects statically, even before the network design has been fully specified.

2 Cocoon by example

In this section, we introduce features of Cocoon by implementing and verifying a variant of the enterprise

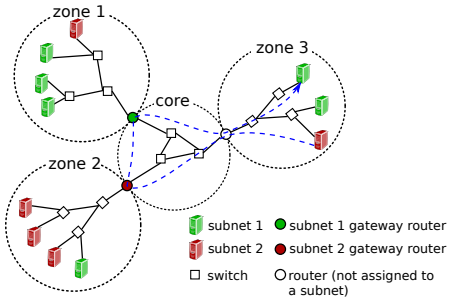


Figure 2: Example enterprise network.

network design described by Sung *et al.* [32], simplified for the sake of presentation. Figure 2 shows the intended network design. Hosts are physically partitioned into operational zones, such as administrative buildings, and grouped by owner into IP subnets symbolized by colors—hosts in each zone are often in the same subnet, but not always. Intra-subnet traffic is unrestricted and isolated by VLAN, but traffic between subnets is subject to an access control policy.

Each operational zone is equipped with a gateway router, which can also be assigned to implement access control for a subnet: Inter-subnet traffic must first traverse the gateway tied to its source subnet followed by the gateway associated with its destination subnet. The details of access control may change as the network runs, but all inter-subnet traffic must always traverse the gateways that implement access control. The path highlighted with a dashed blue line in Figure 2 illustrates traffic from a host in subnet 2 to one in subnet 1.

At a high level, the goals of the network are simple: Group hosts by subnet, allow intra-subnet traffic, and subject inter-subnet traffic to an access control policy (Figure 3a). Our refinement strategy, illustrated in Figure 3, follows the hierarchical struture of the network: the first refinement (Figure 3b) splits the network into operational zones connected via the core network, and distributes access control checks across gateway routers. The second and third refinements detail L2 switching inside zones and the core respectively (Figure 3c and d). We formalize these refinements in the Cocoon language, introducing language features along the way. Figure 4 shows the high-level specification that matches Figure 3a.

Roles The main building blocks of Cocoon specifications are *roles*, which specify arbitrary network entities: hosts, switches, routers, *etc.* A role accepts a packet, possibly modifies it and forwards to zero or more other roles. Roles are *parameterized*, so a single role can specify a set of similar entities, allowing a large network to be modeled with a few roles. An *instance* of the role corresponds to a concrete parameter assignment. A role has an associated *characteristic function*, which determines the set of its instances: Given a parameter assignment,

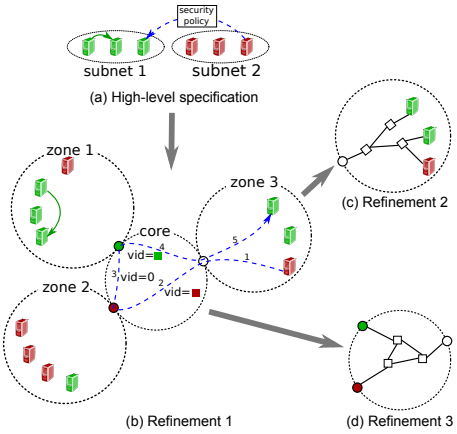


Figure 3: Refinement plan for the running example.

```
1 typedef uint<32> IP4
2 typedef uint<12> vid_t
3 typedef struct {
4   vid_t vid,
5   IP4 srcIP,
6   IP4 dstIP
7 } Packet
8
9 function cHost(IP4 addr): bool
10 function cSubnet(vid_t vid): bool
11 function acl(Packet p): bool
12 function ip2subnet(IP4 ip): vid_t
13 assume(IP4 addr) cHost(addr)=>cSubnet(ip2subnet(addr))
14 function sameSubnet(vid_t svid, vid_t dvid): bool =
15   svid == dvid;
16
17 role HostOut[IP4 addr] | cHost(addr) =
18   let vid_t svid = ip2subnet(pkt.srcIP);
19   let vid_t dvid = ip2subnet(pkt.dstIP);
20   filter addr == pkt.srcIP;
21   filter sameSubnet(svid, dvid) or acl(pkt);
22   filter cHost(pkt.dstIP);
23   send HostIn[pkt.dstIP]
24
25 role HostIn[IP4 addr] | cHost(addr) = filter false
```

Figure 4: High-level specification of the running example.

the characteristic function returns true if and only if the corresponding instance of the role exists in the network.

We use separate roles to model input and output ports of hosts and switches. The input port specifies how the host or switch modifies and forwards packets. The output port specifies how the network handles packets generated by the host. Our high-level specification introduces `HostIn` and `HostOut` roles, which model the input and output ports of end hosts. Both roles are parameterized by the IP address of the host (parameters are given in square brackets in lines 17 and 26), with the characteristic function `cHost` (expression after the vertical bar), declared in line 9.

Policies A role's *policy* specifies how its instances modify and forward packets. Cocoon's policy language is inspired by the Frenetic family of languages [12]: complex policies are built out of primitive policies using sequential and parallel composition. Primitive policies include filtering packets based on header values, updating header fields, and sending packets to other roles.

The `HostOut` policy in lines 18–23 first computes sub-

net IDs of the source and destination hosts and stores them in *local variables*, explained below. Next, it performs two security checks: (1) filter out packets whose source IP does not match the IP address of the sending host (the `filter` operator drops packets that do not satisfy the filter condition) (line 21), and (2) drop packets sent across subnets that violate the network's security policy (line 21). Line 22 drops packets whose destination IP does not exist on the network. All other packets are sent to the input port of their destination host in line 23.

The `send` policy on line 23 is a key abstraction mechanism of Cocoon. It can forward the packet to any instance of any role. While a `send` may correspond to a single hop in the network's data plane, e.g., sending from an input to an output port of the same switch or between two connected ports of different switches, it can also forward to instances without a direct connection to the sender, thus abstracting multiple hops through network nodes not yet introduced at the current refinement level. Cocoon's final specification may only contain the former kind of `send`'s, which can be compiled directly to switch flow tables.

The `HostIn` policy in line 25 acts as a packet sink, dropping all packets delivered to it. Any packets sent by the host in response to previously-received packets are interpreted as new packets entering the network.

Variables The `HostOut` role illustrates three kinds of variables available to a policy: (1) the `pkt` variable, representing the packet processed by the role, which is passed as an implicit argument to each role and can be both read and modified by the policy; (2) read-only role parameters; and (3) local variables that store intermediate values while the role is processing the packet.

Functions Functions are pure (side-effect free) computations used in specifying the set of role instances and defining policies. Function declarations can provide an explicit definition with their body (e.g., `sameSubnet` in Figure 4), or only a signature (e.g., `cHost`, `cSubnet`, `acl` and `ip2subnet`) without a definition. In the latter case, the body of the function can be defined by subsequent refinements, or the body can be dynamically defined and updated at run time, making the function a *runtime-defined function* (RDF).

Our top-level specification introduces four RDFs: `cHost` (discussed above); `cSubnet`, a characteristic function of the set of IP subnets (each subnet is given a unique identifier); `ip2subnet`, which maps end hosts to subnet IDs based on the IP prefix; and `acl`, the network security policy, which filters packets by header fields.

RDFs are a crucial part of Cocoon's programming model. They separate static network design from its runtime configuration. In our example, explicit definitions of RDFs are immaterial to the overall logic of the network operation—making those functions runtime-defined enables specifying the network design along with *all* possible runtime configurations it can produce.

At run time, RDFs serve as the network configuration interface. For example, by redefining RDFs in Figure 4, the operator can introduce new hosts and subnets, update the security policy, *etc.* However, not all possible definitions correspond to well-formed network configurations. In order to eliminate inconsistent definitions, Cocoon relies on *assumptions*.

Assumptions Assumptions constrain the range of possible instantiations of functions—both explicit instantiations in a later refinement and runtime instantiations in the case of RDFs—without fixing a concrete instantiation. Consider the `ip2subnet()` function, which maps end hosts to subnets. We would like to restrict possible definitions of this function to map valid end host IP addresses to valid subnet IDs. Formally,
$\forall \texttt{addr}.\texttt{cHost}(\texttt{addr}) \Rightarrow \texttt{cSubnet}(\texttt{ip2subnet}(\texttt{addr}))$
Line 13 states this assumption in the Cocoon language.

In general, Cocoon assumptions are in the fragment of first-order logic of the form $\forall \texttt{x}_1 \ldots \texttt{x}_i.\texttt{F}(\texttt{x}_1 \ldots \texttt{x}_i)$, where `F` is a quantifier-free formula using variables `x`. This fragment has been sufficiently expressive for the systems we examine and allows for efficient verification.

Until a function is given a concrete definition, Cocoon assumes that it can have any definition that satisfies all its assumptions. Refinements are verified with respect to these assumptions. When the function is defined in a later refinement step, Cocoon statically verifies that the definition satisfies its assumptions. Cocoon performs this verification at run time for RDFs.

Refinements A refinement replaces one or more roles with a more detailed implementation. It provides a new definition of the refined role and, typically, introduces additional roles, so that the composition of the refined role with the new roles behaves according to the original definition of the role.

Consider Refinement 1 in Figure 3b, which introduces zone routers. It refines the `HostOut` role to send packets to the local zone router, which sends them via the two gateway routers to the destination zone router and finally the destination host. The routers are modeled by four new roles, which model the two router ports facing core and zone networks (Figure 5).

Figure 7 illustrates this refinement, focusing on roles. Blue arrows show the packet path that matches the path in Figure 3b. Solid arrows correspond to hops between different network nodes (routers or hosts); dashed arrows show packet forwarding between incoming and outgoing ports of the same router. Both types of hops are expressed using the `send` operation in Figure 6, which shows the Cocoon specification of this refinement.

Line 55 in Figure 6 shows the refined specification of `HostOut`, which sends the packet directly to the destina-

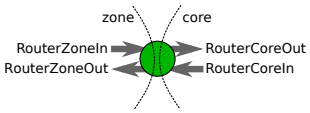


Figure 5: Router ports and corresponding roles.

tion only if it is on the same IP subnet and inside the same zone (line 62); otherwise it sends to the local zone router (line 65). Router roles forward the packet based on its source and destination addresses. They encode the current path segment in the VLAN ID field of the packet, setting it to the source subnet ID when traveling to the source gateway (segments 1 and 2), 0 when traveling between source and destination gateways (segment 3), and destination subnet ID in segments 4 and 5. The security check is now split into two: The `aclSrc()` check performed by the outgoing gateway (lines 19 and 42) and the `aclDst()` check performed by the incoming gateway of the destination subnet (line 31). The assumption in line 10 guarantees that a conjunction of these two checks is equivalent to the global security policy expressed by the `acl()` function. This assumption enables Cocoon to establish correctness of the refinement without getting into the details of the network security policy, which may change at run time.

Subsequent refinements detail the internal structure of core and zone networks. We only show the core network refinement (Figure 3c). For simplicity, Figure 8 specifies the core switching fabric as a single Ethernet switch with switch port number *i* connnected to zone *i* router. This simplification is localized to a single refinement: As the network outgrows the single-switch design, the network programmer can later revise this refinement without affecting the high-level specification or other refinements.

The refined `RouterCoreOut` role (Figure 8, line 2) forwards packets to the core switch rather than directly to the destination router. The core switch input port (line 4) determines the destination router based on the VLAN ID and destination IP address (as one final simplification, we avoid reasoning about IP to MAC address mapping by assuming that switches forward packets based on IP addresses) and forwards the packet via the corresponding output port.

Putting it all together Figure 9 shows the final step in specifying our example network: adding the physical network elements (hosts, switches, and routers). Recall that a role can model any network entity, from an individual interface to an entire network segment. For the Cocoon compiler to generate flow tables, it needs to know how roles combine to form each data-plane element. This is achieved using declarations in lines 1–5, which introduce parameterized hosts and switches (Cocoon currently models routers as switches), specified in terms or their input/output port pairs. A port pair can represent multiple physical ports of the switch. We omit a detailed description of `host` and `switch` constructs, as these are incidental to the main ideas of this work.

Other language features Cocoon supports multicast forwarding using the `fork` construct. For example,

```
fork(uint<16> port|port>0 and port<n())
  send SwitchOut[port]
```

spawns a parallel copy of the `send` statement for each assignment to the `port` variable satisfying the fork condition (expression after the vertical bar). Each parallel thread operates on a private copy of the packet. Note that `n()` can be an RDF, in which case the number forked is determined at run time.

Underspecified behaviors can be expressed using nondeterminism. In the following snippet

```
havoc pkt.dstIP; assume pkt.dstIP != pkt.srcIP
```

the `havoc` statement non-deterministically picks a value for the `dstIP` field of the packet; the `assume` statement constrains the possible choices. Non-determinism is only allowed in high-level specifications and cannot occur in the final, most detailed, definition of any role.

3 Refinement-based verification

We informally present the semantics of Cocoon specifications, the kinds of correctness guarantees that can be established through refinement-based verification, and the design of Cocoon verification tools. See Appendix A for a more formal presentation.

Semantics We start with assigning semantics to roles. Let `Pkt` be the set of all possible packets, and `Loc` be the set of *locations*, where each location identifies a unique role instance in a Cocoon specification. We define the set of *located packets* $\texttt{LPkt} = \{(\texttt{p},\texttt{l}) \mid \texttt{p} \in \texttt{Pkt}, \texttt{l} \in \texttt{Loc}\}$.

We define semantics of a role `R` as a partial function $[\![\texttt{R}]\!] : \texttt{LPkt} \nrightarrow 2^{2^{\texttt{LPkt}}}$ from a located packet to a set of sets of located packets. If Cocoon were deterministic, $[\![\texttt{R}]\!]$ would just return the set of packets produced by role `R` on a given located packet. To model nondeterminism in the semantics, $[\![\texttt{R}]\!]$ returns all possibilities of such sets of packets, thus forming a set of sets of packets.

We define refinement relation $\sqsubseteq$ over roles:

Definition 1 (Role refinement)**.** Role $\hat{\texttt{R}}$ refines role `R` ($\hat{\texttt{R}} \sqsubseteq \texttt{R}$) iff $\hat{\texttt{R}}$ and `R` have identical parameter lists and characteristic functions and

$$\forall \texttt{p} \in \texttt{Domain}([\![\texttt{R}]\!]).[\![\hat{\texttt{R}}]\!](\texttt{p}) \subseteq [\![\texttt{R}]\!](\texttt{p}). \qquad (1)$$

A Cocoon program defines a sequence of specifications, where a specification consists of a set of roles. Each `refine{...}` block introduces a new specification obtained from the previous specification by providing new implementations for some of the roles and introducing new roles.

Next, we informally introduce the `inline()` operation, which takes a role `R` and a set of roles $\{\texttt{P}_1 \ldots \texttt{P}_k\}$ and recursively inlines the implementation of $\texttt{P}_i$ in `R`

```
refine HostOut {
  typedef uint<16> zid_t
  function cZone(zid_t zid): bool
  function zone(IP4 addr): zid_t
  assume (IP4 addr) cHost(addr) => cZone(zone(addr))
  function gwZone(vid_t vid): zid_t
  assume (vid_t vid) cSubnet(vid) => cZone(gwZone(vid))
  function aclSrc(Packet p): bool
  function aclDst(Packet p): bool
  assume (Packet p) acl(p) == (aclSrc(p) and aclDst(p))
  assume (vid_t vid) cSubnet(vid) => (vid != 0)

  role RouterZoneIn[zid_t zid] | cZone(zid) =
    let vid_t dvid = ip2subnet(pkt.dstIP);
    let vid_t svid = pkt.vid;
    filter cSubnet(dvid);
    if dvid != svid and gwZone(svid) == zid then {
      pkt.vid := 0;
      filter aclSrc(pkt)
    };
    send RouterCoreOut[zid]

  role RouterZoneOut[zid_t zid] | cZone(zid) =
    filter cHost(pkt.dstIP) and zone(pkt.dstIP) == zid;
    pkt.vid := 0;
    send HostIn[pkt.dstIP]

  role RouterCoreIn[zid_t zid] | cZone(zid) =
    let vid_t dvid = ip2subnet(pkt.dstIP);
    if pkt.vid == 0 then {
      filter aclDst(pkt);
      pkt.vid := dvid;
      if zone(pkt.dstIP) == zid then
        send RouterZoneOut[zid]
      else
        send RouterCoreOut[zid]
    } else if pkt.vid == dvid then {
      send RouterZoneOut[zid]
    } else {
      let vid_t svid = pkt.vid;
      pkt.vid := 0;
      filter aclSrc(pkt);
      send RouterCoreOut[zid]
    }
  role RouterCoreOut[zid_t zid]|cZone(zid)=
    if pkt.vid == 0 then {
      filter cSubnet(ip2subnet(pkt.dstIP));
      send RouterCoreIn[gwZone(ip2subnet(pkt.dstIP))]
    } else if pkt.vid != ip2subnet(pkt.dstIP) then {
      send RouterCoreIn[gwZone(ip2subnet(pkt.srcIP))]
    } else {
      filter cZone(zone(pkt.dstIP));
      send RouterCoreIn[zone(pkt.dstIP)]
    }
  role HostOut[IP4 addr] | cHost(addr) =
    let vid_t svid = ip2subnet(pkt.srcIP);
    let vid_t dvid = ip2subnet(pkt.dstIP);
    filter addr == pkt.srcIP;
    filter pkt.vid == 0;
    if svid==dvid and zone(pkt.dstIP)==zone(addr)then
    { filter cHost(pkt.dstIP);
      send HostIn[pkt.dstIP]
    } else {
      pkt.vid := ip2subnet(addr);
      send RouterZoneIn[zone(addr)]
    }
}
```

Figure 6: Refinement 1.

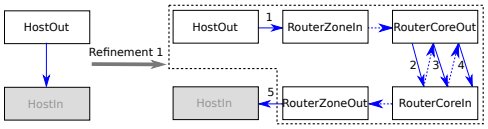


Figure 7: Refinement 1: the `HostOut` role is refined by introducing four router roles. The path from `HostOut` to `HostIn` is decomposed into up to 5 segments.

```
refine RouterCoreOut {
  role RouterCoreOut[zid_t zid] |cZone(zid) =
      send CoreSwitchIn[zid]
  role CoreSwitchIn[uint<16> port] | cZone(port) =
    if pkt.vid == 0 then {
      filter cSubnet(ip2subnet(pkt.dstIP));
      send CoreSwitchOut[gwZone(ip2subnet(pkt.dstIP))]
    } else if pkt.vid != ip2subnet(pkt.dstIP) then {
      send CoreSwitchOut[gwZone(ip2subnet(pkt.srcIP))]
    } else {
      filter cZone(zone(pkt.dstIP));
      send CoreSwitchOut[zone(pkt.dstIP)]
    }
  role CoreSwitchOut[uint<16> port] | cZone(port) =
    send RouterCoreIn[port]
}
```

Figure 8: Refinement 2.

whenever $\mathtt{R}$ sends to $\mathtt{P_i}$. Consider the refinement in Figure 7. When verifying this refinement, we would like to prove that the refined `HostOut` role combined with the newly introduced router roles is equivalent to the original `HostOut` role on the left. This combined role, indicated with the dashed line in Figure 7, is computed as $\mathtt{inline}(\mathtt{HostOut}, \{\mathtt{RouterZoneIn}, \ldots\})$.

We extend the refinement relation to specifications:

Definition 2 (Specification refinement)**.** Let $\mathtt{S} = \{\mathtt{R_1}, \ldots, \mathtt{R_n}\}$ and $\hat{\mathtt{S}} = \{\mathtt{R'_1}, \ldots, \mathtt{R'_n}, \mathtt{P_1}, \ldots, \mathtt{P_k}\}$ be two specifications, such that roles $\mathtt{R_i}$ and $\mathtt{R'_i}$ have identical names, parameter lists and characteristic functions. $\hat{\mathtt{S}}$ refines $\mathtt{S}$ ($\hat{\mathtt{S}} \sqsubseteq \mathtt{S}$) iff $\forall \mathtt{i} \in [1..\mathtt{n}]$. $\mathtt{inline}(\mathtt{R'_i}, \{\mathtt{P_1}, \ldots, \mathtt{P_k}\}) \sqsubseteq \mathtt{R_i}$.

```
host Host[IP4 addr]((HostIn, HostOut))
switch ZoneRouter[zid_t zid](
    (RouterZoneIn,RouterZoneOut),
    (RouterCoreIn,RouterCoreOut))
switch CoreSwitch[]((CoreSwitchIn, CoreSwitchOut))
```

Figure 9: Declaring physical network elements: hosts and switches.

The following proposition is the foundation of Cocoon's compositional verification procedure:

Proposition 1. The $\sqsubseteq$ relation is transitive.

Hence, we can prove that the final specification is a refinement of the top-level specification by proving stepwise refinements within a chain of intermediate specifications.

We encode the problem of checking the refinement relation between roles into a model checking problem and use the Corral model checker [19] to solve it. We chose Corral over other state-of-the-art model checkers due to its expressive input language, called Boogie [20], which enables a straightforward encoding of Cocoon specifications. Given roles $\mathtt{R}$ and $\hat{\mathtt{R}}$, we would like to check property (1) or, equivalently, $\neg(\exists \mathtt{p}, \mathtt{p'}.\mathtt{p'} \in [\![\hat{\mathtt{R}}]\!](\mathtt{p}) \wedge \mathtt{p'} \notin [\![\mathtt{R}]\!](\mathtt{p}))$ (to simplify presentation, we assume that roles are unicast, i.e., output exactly one packet). We encode this property as a Boogie program:

$$\mathtt{p'} := \mathtt{proc}\hat{\mathtt{R}}(\mathtt{p}); \ \mathtt{assert}(\mathtt{proc}\mathtt{R}(\mathtt{p}, \mathtt{p'}))$$

Here, $\mathtt{proc}\hat{\mathtt{R}}(\mathtt{p})$ is a Boogie procedure that takes a located packet $\mathtt{p}$ and non-deterministically returns one of

possible outputs of $\hat{\mathtt{R}}$ on this packet; $\mathtt{procR}(\mathtt{p}, \mathtt{p}')$ returns `true` iff $\mathtt{p}' \in \mathtt{R}(\mathtt{p})$. We use Boogie's `havoc` construct to encode nondeterminism. We encode Cocoon assumptions as Boogie axioms, and characteristic functions of roles as procedure preconditions [20]. Violation of property (1) triggers an assertion violation in this program.

Corral is a bounded model checker, i.e., it only detects assertion violations that occur within a bounded number of program steps. We sidestep this limitation by bounding the maximal number of network hops introduced by each refinement. This is a natural restriction in network verification, as any practical network design must bound the number of hops through the network. We introduce a counter incremented by every `send` and generate an error when it exceeds a user-defined bound, which is also used as a bound on the number of program steps explored by Corral. Coincidentally, this check guarantees that refinements do not introduce forwarding loops.

Verifying path properties Cocoon's refinement-based verification operates on a single role at a time and, after the initial refinement, does not consider global forwarding behavior of the network. Importantly, however, it guarantees that all such behaviors are preserved by refinements, specifically, a valid refinement can only modify a network path by introducing intermediate hops into it; however, it cannot modify paths in any other way, add or remove paths.

This invariant can be exploited to dramatically speed up conventional property-based data plane verification. Consider, for example, the problem of checking pairwise reachability between all end hosts. Cocoon guarantees that this property holds for the network implementation if and only if it holds for its high-level specification. Often, the high-level specification is simple enough that the desired property obviously holds for it. If, however, the user does not trust the high-level specification, they can apply an existing network verification tool such as NetKAT, HSA, or Veriflow to it. In Section 6.3, we show that such verification can be performed much more efficiently than checking an equivalent property directly on the detailed low-level implementation.

Limitations Because Cocoon specifications describe how individual packets are forwarded, it cannot verify properties related to multiple packets such as stateful network behaviors induced by say stateful firewalls. This limitation is shared by virtually all current network verification tools, which verify data plane *snapshots*.

However, stateful networks *can* be built on top of Cocoon by encapsulating dynamic state inside RDFs. For example, a stateful firewall specification may include a function that determines whether a packet must be blocked by the firewall. This function is computed by an external program, potentially based on observed packet history. Cocoon can enforce statically defined invariants over such functions. For example, with multiple firewalls, it can enforce rule set consistency and ensure that each entering packet is inspected by one firewall.

Assumption checker Cocoon's dynamic assumption checker encodes all function definitions and assumptions into an SMT formula and uses the Z3 SMT solver [7] to check the validity of this formula.

4 Compiler

The Cocoon compiler proactively compiles specifications into switch flow tables; it currently supports OpenFlow and P4 backends. Due to space limitations, we only describe the OpenFlow backend.

The OpenFlow backend uses NetKAT as an intermediate representation and leverages the NetKAT compiler to generate OpenFlow tables during the final compilation step. Compilation proceeds in several phases. The first phase computes the set of instances of each role by finding all parameter assignments satisfying the characteristic function of the role with the help of an SMT solver. During the second phase, we specialize the implementation of each role for each instance by inlining function calls and substituting concrete values for role parameters.

The third phase constructs a network topology graph, where vertices correspond to hosts and switches, while edges model physical links. To this end, the compiler statically evaluates all instances whose roles are listed as outgoing ports in `host` and `switch` specifications and creates an edge between the outgoing port and the incoming port it sends to. The resulting network graph is used in an emulator (Section 6).

During the fourth phase, instances that model input ports of switches are compiled to a NetKAT program. This is a straightforward syntactic transformation, since NetKAT is syntactically and semantically close to the subset of the Cocoon language obtained after function inlining and parameter substitution. During the final compilation phase, the NetKAT compiler converts the NetKAT program into OpenFlow tables to be installed on network switches.

The resulting switch configuration handles all packets inline, without ever forwarding them to the controller. An alternative compilation strategy would be to forward some of the packets to the controller, which would enable more complex forms of packet processing that are not supported by the switch.

At run time, the Cocoon compiler translates network configuration updates into updates to switch flow tables. Recompiling the entire data plane on every reconfiguration is both inefficient and unnecessary, since most updates only affect a fraction of switches. For instance, a change in the network security policy related to a specific subnet in our running example requires reconfiguring the router assigned to this subnet only. While our current

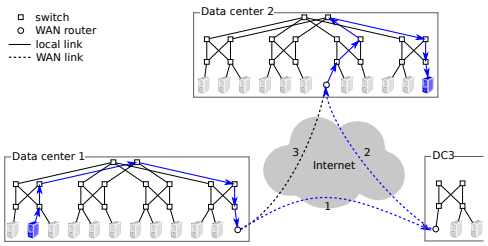


Figure 10: Case study 1: Software-defined WAN. Arrows show an example path between end hosts in different sites.

prototype does not support incremental compilation, it can be implemented as a straightforward extension.

5 Case studies

We show that real-world SDNs can benefit from refinement-based design by implementing and verifying six network architectures using Cocoon. Our case studies cover both mainstream SDN applications such as network virtualization and emerging ones such as software-defined WANs and IXPs. The case studies have multiple sources of complexity including non-trivial routing logic, security constraints, fault recovery; they are hard to implement correctly using conventional tools. We present two studies in detail and briefly outline the remainder.

5.1 Case study 1: Software-defined WAN

We design and verify a software-defined WAN inspired by Google's B4 [14] comprising geographically distributed datacenters connected by wide-area links (Figure 10). It achieves optimal link utilization by sending traffic across multi-hop tunnels dynamically configured by a centralized controller. In Figure 10, some traffic between datacenters 1 and 2 is sent via a tunnel consisting of underutilized links 1 and 2 instead of congested link 3. Cocoon cannot reason about quality-of-service and relies on an external optimizer to choose tunnel configuration; however it can formalize the WAN architecture and enforce routing invariants, which ensure that optimizer configurations deliver packets correctly.

We specify end-to-end routing between end hosts in the WAN, including inter- and intra-datacenter routing. Local and global routing can be specified by different teams and integrated in a common Cocoon specification. Our high-level specification (Figure 11) is trivial: it defines a set of hosts and requires that each packet be delivered to its destination, if it exists:

```
role HostOut[IP4 addr] | cHost(addr) =
  if cHost(pkt.dstIP) then send HostIn[pkt.dstIP]
```

We refine the specification following the natural hierarchical structure of the network: we first decompose the WAN into datacenters (Refinement 1); these are further decomposed into pods (Refinement 2), which are in turn decomposed into three layers of switches (Refinements 3 and 4).

In more detail, Refinement 1 defines global routing

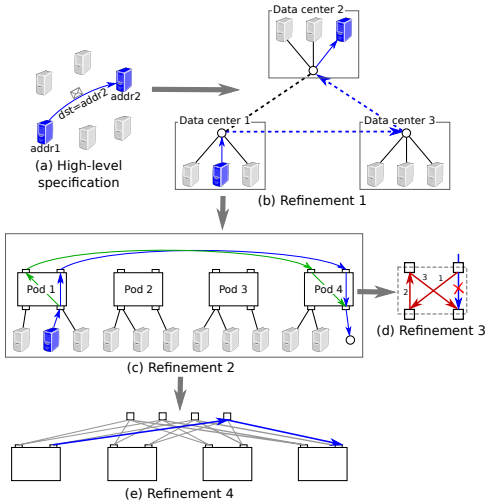


Figure 11: Refinement strategy for case study 1.

and topology. It partitions hosts into subnets, localized within datacenters, and introduces WAN links across datacenters. It formalizes tunnel-based routing using two functions:

```
function tunnel(dcid_t src,dcid_t dst,Packet p): tid_t
function nexthop(tid_t tun,dcid_t dc): dcid_t
```

The former maps a packet to be sent from datacenter `src` to `dst` to ID of the tunnel to forward the packet through. The latter specifies the shape of each tunnel as a chain of datacenters. We define a recursive function `distance(src,dst,tun)`, which computes the number of hops between two datacenters via tunnel `tun`. Correctness of global routing relies on an assumption that tunnels returned by the `tunnel()` function deliver packets to the destination in `k` hops or less:

```
function distance(dcid_t src,dcid_t dst,tid_t tid):u8=
  case {
    src == dst: 8'd0;
    default: distance(nexthop(tid,src),dst,tid) + 1;}
assume(dcid_t src, dcid_t dst, Packet p)
  cDC(src) and cDC(dst)
  => distance(src,dst,tunnel(p)) <= k()
```

where `k` is a user-defined bound on the length of a tunnel and `cDC()` is the characteristic function of the set of datacenters.

Subsequent refinements detail intra-datacenter topology and routing. Specifically, we instantiate a fat-tree topology [1] within each datacenter: other topologies can be specified equally easily. Refinement 2 introduces groups of switches, called *pods*, within the datacenter fabric: each host is connected to a downstream port of a pod, which forwards packets to an upstream port of the same pod, which, in turn, forwards to the destination pod. Pod behavior is underspecified by this refinement: the pod non-deterministically picks one of the upstream ports to send each packet through, giving rise to multiple paths, shown by blue and green arrows. This nondeterminism is resolved by Refinement 3, which decomposes pods into two layers of switches. A bottom-layer

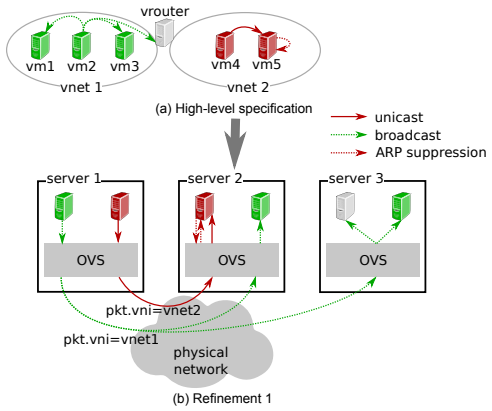


Figure 12: Refinement strategy for case study 2.

switch picks a top-level switch to send to based on the hash of the packet's destination address. Refinement 3 also takes advantage of path redundancy to route packets around failed links. The blue arrow in Figure 11d shows the normal path between top and bottom-layer switches within a pod; red arrows show the backup path taken in case of link failure. Finally, Refinement 4 details packet forwarding between pods via the *core* layer of switches.

5.2 Case study 2: Network virtualization

Network virtualization for multi-tenant datacenters is arguably the most important SDN application today [18]. It combines CPU and network virtualization to offer each client the illusion of running within its own private datacenter. Figure 12a shows the clients' view of the datacenter as a collection of isolated LANs connected only by router nodes that have interfaces on multiple LANs. In reality, client workloads run inside virtual machines (VMs) hosted within physical servers connected by a shared network fabric (Figure 12b). Each server runs an instance of a software SDN switch, OpenVSwitch (OVS) [26], which isolates traffic from different tenants. Packets sent to VMs hosted on remote physical nodes are encapsulated and forwarded to remote OVS instances.

While the basic virtualization scheme is simple, industrial virtualization platforms, such as VMWare NSX [18], have evolved into complex systems, due to numerous configuration options and feature extensions which are hard to understand and use correctly.

In this case study we untangle network virtualization with the help of refinement-based programming. We implement a basic virtualization scheme and a number of extensions in Cocoon. Below we present two example extensions and show how Cocoon separates the specification of various features from their implementation, thus helping users and developers of the framework to understand its operation, while also bringing the benefits of verification to network virtualization.

Service chaining Service chaining modifies the virtual forwarding to redirect packets through a chain of virtual middleboxes. Middlebox chains are formalized by the following RDF, which, based on packet headers and current packet location computes the virtual port to forward the packet to (the destination port or the next middlebox in the chain):

```
function chain(Packet p, VPortId port): VPortId
```

Service chaining required only a minor modification to the high-level specification: instead of forwarding the packet directly to its destination MAC address, we now forward it down the service chain:

```
role VHostOut[VPortId vport] | cVPort(vport) =
  ...
  (*send VHostIn[mac2VPort(vnet, pkt.dstMAC)]*)
  send VHostIn[chain(p, vport)]
```

The implementation of this feature in the refined specification is, however, more complex: upon receiving a packet from a virtual host, OVS uses the `chain()` function to establish its next-hop destination. It then attaches a label to the packet encoding its last virtual location and sends the packet via a tunnel to the physical node that hosts the next-hop destination. OVS on the other end of the tunnel uses the label to determine which virtual host to deliver it to.

Broadcast and ARP suppression Broadcast packets must be delivered to all VMs on the virtual network:

```
role VHostOut[VPortId vport] | cVPort(vport) =
  ...
  if pkt.dstMAC == hffffffffffff(*bcast address*) then
    fork (VPortId vport | vPortVNet(vport) == vnet)
      send VHostIn[vhport.vhost, vhport.vport]
```

This behavior is implemented via two cascading multicasts shown with dashed green arrown in Figure 12b. First, the OVS at the source multicasts the packet to all physical servers that host one or more VMs on the same virtual network. Second, the destination OVS delivers a copy of the packet to each local VM.

The ARP suppression extension takes advantage of the fact that most broadcast traffic consists of Address Resolution Protocol (ARP) requests. When ARP suppression is enabled for a virtual network, Cocoon configures all OVS instances with a local table of IP-to-MAC address mappings, used to respond to ARP requests, locally.

Other extensions we have implemented include a decentralized information flow control model for networks and virtual-to-physical port forwarding.

5.3 Other case studies

Our third case study is a realistic version of the enterprise network, a simplified version of which was used in Section 2 [32]. In addition to features described in Section 2, we accurately model both MAC-based forwarding (within a VLAN) and IP-based forwarding across VLANs, implement support for arbitrary IP topologies that do not assume a central core network, and arbitrary level-2 topologies within each zone. We replace the standard decentralized routing protocols used in the original

design with a SDN controller computing a centralized routing policy. This policy is expressed via RDFs, which are compiled to OpenFlow and installed on all switches.

The fourth case study implements the F10 fault-tolerant datacenter network design. F10 uses a variant of fat tree, extending it with the ability to globally reconfigure the network to reduce performance degradation due to link failures. In a traditional fat tree, a link failure may force the packet to take a longer path through the network, as shown in Figure 11d. F10 avoids this by reconfiguring all potentially affected switches to steer the traffic away from the affected region of the switching fabric. We implement and SDN version of F10, where the reconfiguration is performed by the central controller rather than a decentralized routing protocol.

Case study 5 implements a protocol called sTag [21]—a version of fat tree with source-based routing. The edge router attaches two tags to each packet: an `mTag`, which identifies switch ports to send the packet through at every hop, and a security tag that identifies the sender of the packet. The latter is validated by the last switch in the path, before delivering the packet to the destination.

Our final case study implements the iSDX software-defined Internet exchange point (IXP) architecture [13]. In iSDX, each autonomous system (AS) connected to IXP can define its own routing preferences. These preferences are encoded in the MAC address field of each packet sent from the AS to the IXP. The IXP decodes the MAC address and ensures that AS preferences do not conflict with the BGP routing database.

5.4 Experience with the Cocoon language

We briefly report on our experience with the Cocoon language. We found the language to be expressive, as witnessed by the wide range of network designs covered by our case studies, concise (see Section 6), and effective at capturing the modularity inherent in any non-trivial network. Thus, the language achieves its primary design goal of enabling refinement-based design and verification of a large class of networks. At the same time, we found that by far the hardest part of programming in Cocoon is defining assumptions on RDFs necessary for static verification and runtime checking. Writing and debugging assumptions, such as the one shown in Section 5.1, may be challenging for engineers who are not accustomed to working with formal logic. Therefore, in our ongoing work we explore higher-level language constructs that will offer a more programmer-friendly way to specify assumptions.

6 Implementation and evaluation

We implemented Cocoon in 4,700 lines of Haskell. We implemented, verified, and tested the six case studies described in Section 5 using the Mininet network emulator. Cocoon, along with all case studies, is available under the open source Apache 2.0 license [6].

case study	LOC		#refines	verification time (s)	
	total	high-level		compositional	monolithic
WAN	305	18	6	10	>3600
virtualization	678	97	1	6	6
enterprise	342	50	4	16	>3600
F10	262	52	2	19	57
sTag	283	47	1	2	2
iSDX	190	21	2	3	3

Table 1: Summary of case studies. >3600 in the last column denotes experiments interrupted after one hour timeout.

All experiments in this section were performed on a machine with a 2.6 GHz processor and 128 GB of RAM. We measured single-threaded performance.

Table 1 summarizes our case studies, showing (1) total lines of Cocoon code (LOC) excluding runtime-defined function definitions, (2) lines of code in the high-level specification, (3) number of refinements, (4) time taken by the Cocoon static verifier to verify all refinements in the case study, and (5) time taken to verify the entire design in one iteration. The last column measures the impact of compositional verification: We combine all refinements and verify the combined specification against the high-level specification in one single step.

6.1 Static verification

The results show that Cocoon verifies the static design of complex networks in a matter of seconds with compositional verification being much faster than monolithic verification:[1] a refinement that focuses on a single role has an exponentially smaller state space than the complete specification and is potentially exponentially faster to verify. As we move through the refinement hierarchy, the low-level details of network behavior are partitioned across multiple roles and can be efficiently verified in isolation, while compositionality of refinements guarantees end-to-end correctness of the Cocoon program.

Bug finding We choose 3 (of many) examples from the case studies to show how Cocoon detected subtle bugs early in our designs.

1. Enterprise network: When sending a packet between hosts on different subnets in the same zone, the zone router skipped access control checks at gateway routers (skipping path segments 2-3-4 in Figure 3b). Since this bug only manifests for some topologies and security policies, it is difficult to detect using testing or snapshot verification. The bug was detected early (in refinement 1), before L2 forwarding was introduced.

2. WAN: Consider the packet path in Figure 11d. Our implementation incorrectly sent the packet back to the core after hops 1 and 2, instead of sending it down via hop 3, causing a loop. This bug only manifests in re-

[1] The virtualization and sTag case studies only had one refinement; hence compositional and monolithic verification are equivalent in these examples.

Scale	Hosts	Switches	NetKAT Policy	Flowtable Rules
2	8	11	1,559	830
5	17	23	5,149	3,299
15	47	63	46,094	31,462
25	77	103	151,014	89,216
40	122	163	496,268	212,925

Table 2: Number of hosts, switches, size of NetKAT policy and flowtable rules as a function of network scale.

sponse to a link failure, making it hard to catch by snapshot verification. It was detected only when verifying refinement 3, but the verifier localized the bug in space to the pod component.

3. Virtualization: This bug, discovered when verifying the sole refinement in this case study, is caused by the interplay between routing and security. The specification requires that neither unicast nor multicast packets can be exchanged by blacklisted hosts. The implementation filters out unicast packets at the source OVS; however, multicast packets were filtered at the destination and hence packets delivered to VMs hosted by the same server as the sender bypassed the security check.

For each detected bug, Corral generated two witness traces, which showed how the problematic packet was handled by the abstract and refined implementations respectively. The two traces would differ in either how they modify the packet or in where they forward it.

Our encoding of refinements into Boogie guarantees the absence of false negatives, i.e., Corral does not miss any bugs (modulo defects in Corral itself). However, we have encountered three instances where Corral reported a non-existing bug. In all three cases this was caused by a performance optimization in Corral: by default, it runs the underlying Z3 SMT solver with a heuristic, incomplete, quantifier instantiation strategy. We eliminated these false positives by reformulating some of our assumptions, namely, breaking boolean equivalences into pairs of implications.

6.2 Cocoon vs. NetKAT

The NetKAT decision procedure for program equivalence [11] is the closest alternative to refinement verification. We compare Cocoon against NetKAT on a parameterized model of the enterprise network case study [32]—we configure the network with three operational zones and scale the number of hosts and switches per zone. For an access control policy, we randomly blacklist communication between pairs of hosts. The topology of the operational zones and the router-to-router fabric are built from Waxman graphs. Table 2 summarizes the dimensions of our test network for a sample of scales.

We measure the full verification run time of Cocoon, including the cost of static refinement verification and the cost of checking the assumptions of RDFs. We then perform an equivalent experiment using NetKAT. To this end, we translate each level of Cocoon specification, along with definitions for the RDFs, into NetKAT, and use the NetKAT decision procedure to determine whether the lower-level specification exhibits a subset of the behaviors of the higher-level specification.

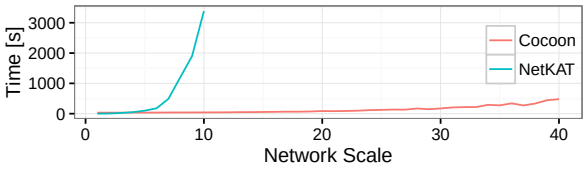


Figure 13: Comparison between Cocoon refinement verification vs. equivalence decision in NetKAT.

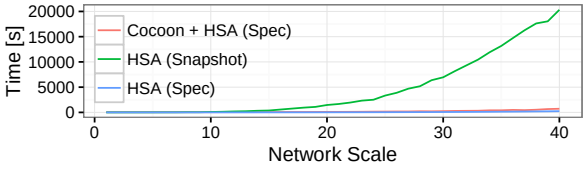


Figure 14: Comparison of high-level specification verification via HSA and Cocoon verification vs. snapshot data-plane verification via HSA.

Figure 13 shows the verification run time in seconds as we increase the network scale. Cocoon verification scales beyond that of NetKAT. Cocoon performs much of the heavy lifting during static verification, taking advantage of all available design-time information captured in refinements, assumptions, and parameterized roles.

6.3 Cocoon + HSA

At present, the easiest way to verify an arbitrary controller application is to verify reachability properties for each of the data-plane configurations it generates. As described in Section 3, Cocoon can accelerate property-based verification: instead of checking path properties on the low-level data-plane configuration, one can check them more efficiently on the top-level Cocoon specification, taking advantage of the fact that such properties are preserved by refinement.

We evaluate this using the Header Space Analysis (HSA) [16] network verifier. Using the same network scenarios as above, we use Frenetic to compile the NetKAT policy (translated from Cocoon specification) into a set of OpenFlow flowtables. We produce the flowtables of the highest- and lowest-level specifications, shown below as Spec and Snapshot, respectively. We then apply HSA by creating the corresponding transfer functions and checking the all-pair reachability property on Spec and Snapshot, and measure the total run time. As can be seen in Figure 14, performing the verification on Spec and leveraging Cocoon's refinement verification results in dramatic improvement in verification performance, so that the cost of Cocoon verification (red line) dominates the cost of HSA applied to the high-level specification (blue line).

7 Related work

SDN controllers often use two-tier design: a controller emits a stream of data-plane configurations. There are many languages and verification techniques for both tiers, and some approaches that abandon the two tiers.

Language design OpenFlow [25] is a data-plane configuration language: Controller frameworks like OpenDaylight [24], Floodlight [9], and Ryu [28] emit OpenFlow commands to update SDN-capable switches. The Frenetic family [2, 10, 22, 29] introduces modular language design; they allow writing controller applications in a general purpose language and compiling to OpenFlow. VeriCon [3], FlowLog [23], and Maple [33] eschew the tiered structure entirely using custom languages that describe network behavior over time.

Cocoon is a data-plane configuration language that inherits its sequential and parallel composition operators from NetKAT [2] while adding roles, refinements, RDFs, and assumptions. These features enable modular verification in addition to the modular program composition of NetKAT and other Frenetic languages.

Data-plane verification SDN verification takes two forms: data-plane verification and controller verification. The former checks that a given set of safety properties (e.g, no black holes or forwarding loops) hold on a given data-plane configuration [15–17]. Hence it must be reapplied to each configuration the controller produces. Further, checking reachability between host pairs scales quadratically with the number of hosts. Verification can be sped up by leveraging symmetries but the problem remains [27]. SymNet [31] is a network modeling language to perform efficient static analysis via symbolic execution of a range of network devices from switches to middleboxes.

Cocoon does not verify network properties directly. Rather, it guarantees that refinements are functionally equivalent, provided dynamically checked assumptions holds on RDF definitions. Often, reachability properties are "obvious" in high-level specifications: They hold by design and are preserved by functional equivalence, and so hold across refinements. If the design is not "obvious", data-plane verification can be applied to the highest level Cocoon specification, which is often dramatically simpler, enabling much faster property verification.

NetKAT [2] is a language with a decision procedure for program equivalence. This enables property verification but can also verify whether one NetKAT program is a correct refinement of another. However, NetKAT verification is not yet suitable for verifying the equivalence of large networks in near-real time. NetKAT lacks the abstractions—namely RDFs and assumptions—that allow some verification to be done statically. NetKAT also lacks other language features that Cocoon provides for stepwise refinement, including parameterized roles, in part because NetKAT is intended as a synthesis target emitted by the Frenetic controller. Cocoon refinements, on the other hand, are human readable, even at scale.

Controller verification VeriCon [3] and FlowLog [23] prove statically that a controller application *always* produces correct data-plane configurations. VeriCon reduces verification to SMT solving, while Flowlog uses bounded model checking. In both cases, scalability is a limiting factor. FlowLog also restricts expressivity to enable verification. NICE [5] uses model checking and symbolic execution to find bugs in Python controller programs but focuses on testing rather than verification.

In contrast, Cocoon statically verifies that refinements are functionally equivalent, but the refinement language is less expressive than either VeriCon or FlowLog—dynamic behavior is excluded, hidden behind RDFs. However, this combination of static and dynamic verification enables much greater scalability (see Section 6), while still providing strong guarantees about arbitrarily complex dynamic behavior hidden in RDFs.

Stepwise refinement Stepwise refinement for programming dates back to Dijkstra [8] and Wirth [34]. In the networking domain, Shankar and Lam [30] proposed an approach to network protocol design via stepwise refinement. Despite its promise, refinement-based programming has had limited success in mainstream software engineering because: (1) developing formal specifications for non-trivial software systems is hard, (2) formalizing module boundaries for compositional verification is equally hard; even well designed software systems modules make implicit assumptions, and (3) verifying even simple software modules automatically is hard.

8 Conclusions

Our key discovery is that the factors that impede refinement based software engineering are not roadblocks to refinement-based *network* programming. First, even complex networks admit relatively simple high-level specifications. Second, boundaries between different network components admit much cleaner specifications than software interfaces. Finally, once formally specified, network designs can be efficiently verified.

Cocoon can be seen as both a *design assistant* and a *proof assistant*: by imposing the refinement-based programming discipline on the network designer, it enforces more comprehensible designs that are also amenable to efficient automatic verification.

Although we apply our techniques to SDNs, we expect them to be equally applicable to traditional networks. In particular, stepwise refinement may help find bugs in potentially complex interactions between mechanisms such as VLANs and ACLs, and check that forwarding state matches the assumptions made in the specifications.

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[2] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic Foundations for Networks. In *POPL*, 2014.

[3] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. VeriCon: Towards Verifying Controller Programs in Software-Defined Networks. In *PLDI*, 2014.

[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3), July 2014.

[5] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *NSDI*, 2012.

[6] Cocoon website. `https://github.com/ryzhyk/cocoon`.

[7] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS/ETAPS*, 2008.

[8] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3), Sept. 1968.

[9] Floodlight. `http://www.projectfloodlight.org/`.

[10] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ICFP*, 2011.

[11] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson. A Coalgebraic Decision Procedure for NetKAT. In *POPL*, 2015.

[12] Frenetic. `http://frenetic-lang.org/`.

[13] A. Gupta, R. MacDavid, R. Birkner, M. Canini, N. Feamster, J. Rexford, and L. Vanbever. An Industrial-scale Software Defined Internet Exchange Point. In *NSDI*, 2016.

[14] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-Deployed Software Defined WAN. In *SIGCOMM*, 2013.

[15] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *NSDI*, 2013.

[16] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *NSDI*, 2012.

[17] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *NSDI*, 2013.

[18] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network Virtualization in Multi-tenant Datacenters. In *NSDI*, 2014.

[19] A. Lal, S. Qadeer, and S. K. Lahiri. A Solver for Reachability Modulo Theories. In *CAV*, 2012.

[20] K. R. M. Leino. This is Boogie 2, June 2008. Manuscript KRML 178 `http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf`.

[21] N. Lopes, N. Bjørner, N. McKeown, A. Rybalchenko, D. Talayco, and G. Varghese. Automatically verifying reachability and well-formedness in P4 Networks. Technical Report MSR-TR-2016-65, Microsoft Research, Sept. 2016.

[22] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A Compiler and Run-time System for Network Programming Languages. In *POPL*, 2012.

[23] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi. Tierless Programming and Reasoning for Software-Defined Networks. In *NSDI*, 2014.

[24] OpenDaylight. `https://www.opendaylight.org/`.

[25] The OpenFlow protocol. `https://www.opennetworking.org/sdn-resources/openflow`.

[26] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *NSDI*, 2015.

[27] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese. Scaling Network Verification Using Symmetry and Surgery. In *POPL*, 2016.

[28] Ryu. `https://osrg.github.io/ryu/`.

[29] C. Schlesinger, M. Greenberg, and D. Walker. Concurrent NetCore: From Policies to Pipelines. In *ICFP*, 2014.

[30] A. U. Shankar and S. S. Lam. Construction of Network Protocols by Stepwise Refinement. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, 1990.

[31] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. SymNet: scalable symbolic execution for modern networks. In *SIGCOMM*, 2016.

[32] Y.-W. E. Sung, S. G. Rao, G. G. Xie, and D. A. Maltz. Towards Systematic Design of Enterprise Networks. In *CoNEXT*, 2008.

[33] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *SIGCOMM*, 2013.

[34] N. Wirth. Program Development by Stepwise Refinement. *Commun. ACM*, 14(4), Apr. 1971.

[35] Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. Answering Why-Not Queries in Software-Defined Networks with Negative Provenance. In *HotNets*, 2013.

A Syntax and semantics of Cocoon

A.1 Syntax

The syntax of the Cocoon system is given in Fig. 15. Let `Id` be the set of identifiers, `Pkt` the set of packets and `Val` the set of values.

We suppose the existence of a countable set of identifiers, or variable names. Values comprise booleans `true` and `false`, integers, tuples, and records of type id, written $id\{v\}$. Expressions comprise standard negation, binary operators $\otimes$, projection of fields $e.id$, construction of records $id\{e\}$, function call $id(e)$, built-in function calls $id!(e)$, variable call id, tuple construction $(e,\ldots,e)$, and call to the current packet being processed `pkt`. The semantics of a built-in function $id!$ is given by $[\![id!]\!] \in \texttt{Val} \to \texttt{Val}$.

Statements allow filtering, which stops the computation if e does not evaluate to `true`. Assumptions are slightly different in that they are not executable, but can be refined only if e evaluates to `true`. Packet fields can be assigned explicitly with the $:=$ construct, or assigned

integers	n	
identifier	id	
	ids	$= id \mid id, ids$
arguments	$args$	$= \tau\ id \mid \tau\ id, args$
case body	$cbod$	$= \cdot \mid e_1 : e_2; cbod$
value	v	$= \texttt{true} \mid \texttt{false} \mid n \mid id\{v\}$
		$(v,\ldots,v)$
expression	e	$= \texttt{not}\ e \mid e_1 \otimes e_2 \mid e\texttt{.}id$
		$\mid id\,\texttt{(}e\texttt{)} \mid id\texttt{!(}e\texttt{)} \mid id\{e\}$
		$\mid \texttt{pkt} \mid id$
		$\mid \texttt{case}\ \{cbod\texttt{; default:}e\texttt{;}\}$
		$\mid \texttt{(}e,\ldots,e\texttt{)}$
type specs	τ	$= \texttt{uint <}n\texttt{>} \mid \texttt{bool} \mid id \mid \texttt{[}\tau\texttt{;}n\texttt{]}$
		$\mid \texttt{struct}\ \{args\}$
statement	a	$= \texttt{filter}\ e \mid \texttt{assume}\ e$
		$\mid \texttt{pkt.}id\texttt{:=}e \mid \texttt{havoc pkt.}id$
		$\mid \texttt{if}\ e\ \texttt{then}\ a_1$
		$\mid \texttt{if}\ e\ \texttt{then}\ a_1\ \texttt{else}\ a_2$
		$\mid \texttt{let}\ \tau\ id\ =\ e$
		$\mid \texttt{send}\ id\texttt{[}e\texttt{]}$
		$\mid \texttt{fork}\ \texttt{(}args\ \texttt{|}\ e\texttt{)}\ as$
	as	$= f \mid a, as$
role constraints	cs	$= \cdot \mid \texttt{|}e \mid \texttt{/}e \mid \texttt{|}e\texttt{/}e$
declaration	d	$= \texttt{typedef}\ \tau\ id$
		$\mid \texttt{function}\ id\,\texttt{(}args\texttt{)} : \tau$
		$\mid \texttt{function}\ id\,\texttt{(}args\texttt{)} : \tau = e$
		$\mid \texttt{role}\ id\texttt{[}args\texttt{]}\ cs\ \ = as$
		$\mid \texttt{assume}\ \texttt{(}args\texttt{)}\ e$
	ds	$= d \mid d, ds$
refinement	r	$= \texttt{refine}\ ids\ ds$
spec	$spec$	$= r \mid r, spec$

Figure 15: Cocoon Syntax.

to a nondeterministic value using `havoc`. Statements allow standard conditionals and let-bindings. Finally, a statement can send a packet to another role $id[e]$, or fork to multicast a packet across all variables $args$ satisfying condition e.

Declarations contain function definitions, both without a body to be refined later on or become user-defined functions, and with a body when defined explicitly. Declarations also contain role definitions: a role is parameterized by some arguments, and is only valid if some constraints on those arguments ($|\ e$) and on the incoming packets ($/\ e$) are true. Finally, assumptions allow restricting the future definitions of declared functions, both in future refinements and as user-defined functions.

Note that although types are part of the syntax, we drop them in the semantics to simplify notations.

A.2 Semantics

We give a denotational semantics of Cocoon. The semantics of expressions, statements and declarations is

given in terms of:

- a packet $p \in \texttt{Pkt}$, a record of type `Pkt`;
- a local environment $\sigma \in (\texttt{Id} \rightharpoonup \texttt{Val})$, a partial function from identifiers to values, comprising `let`-defined variables; let `Env` be the set of local environments;
- a set of possible environments of functions ϕ. Functions take one argument (which can be a tuple). Each function's denotational semantics is a (mathematical) function from a pair (v,p) to a value v. Each possible environment of functions is a partial function from identifiers to such denotational semantics. The set ϕ is a set of such possible environments, representing all the possible function definitions. If Φ represents the set of all the sets of possible environments of functions, we thus have
$$\Phi = \mathscr{P}(\texttt{Id} \rightharpoonup (\texttt{Val} \times \texttt{Pkt}) \to \texttt{Val})$$
This enables the modelling of the nondeterminism introduced by functions defined only as signatures, and possibly restrained by assumptions;
- an environment of roles ρ, a partial function from identifiers to role semantics, where each role semantics is a (mathematical) function from a pair (v,p) to a set of sets of pairs (p,σ). Each set of pairs (p,σ) represents one possible execution, possibly returning multiple packets in the case of multicasting. We model nondeterminism by having the semantics return a set of those possible executions. Thus sets of sets enable the modeling of both nondeterminism and multicasting. If P represents the possible environments of roles, we thus have
$$P = (\texttt{Id} \rightharpoonup \texttt{Val} \times \texttt{Pkt} \to \mathscr{P}(\mathscr{P}(\texttt{Pkt} \times \texttt{Env})))$$

A.2.1 Semantics of expressions

The semantics of expressions is given in Figure 16, in terms of a triple (p,σ,ϕ). Expressions are nondeterministic, and thus their semantics is a set of possible output values.

Most of the semantics is standard. Note that the only nondeterminism is introduced by a function call $id(e)$. Functions are defined in the environment ϕ, while built-in functions $id!$ have their own semantics. The call `pkt` just returns the current packet p in all cases.

A.2.2 Semantics of statements

The semantics of statements is given in Figure 17, in terms of a quadruplet (ρ,σ,ϕ,ρ), and returns a set of sets of pairs (p,σ) to model both nondeterminism and multicasting.

$$\begin{aligned}
[\![e]\!](p,\sigma,\phi) &\in \mathscr{P}(\texttt{Val}) \\
[\![v]\!](p,\sigma,\phi) &= \{v\} \\
[\![\texttt{not } e]\!](p,\sigma,\phi) &= \{\neg\, v \mid v \in [\![e]\!]\} \\
[\![e_1 \otimes e_2]\!](p,\sigma,\phi) &= \{v_1 \otimes v_2 \mid v_1 \in [\![e_1]\!], v_2 \in [\![e_2]\!]\} \\
[\![e.id]\!](p,\sigma,\phi) &= \{v.id \mid v \in [\![e]\!]\} \\
[\![id(e)]\!](p,\sigma,\phi) &= \{f(id)(v,p,\phi) \mid v \in [\![e]\!], f \in \phi\} \\
[\![id!(e)]\!](p,\sigma,\phi) &= \{[\![id!]\!](v) \mid v \in [\![e]\!]\} \\
&\quad \text{where } [\![id!]\!] \in \texttt{Val} \to \texttt{Val} \\
[\![id\{e\}]\!](p,\sigma,\phi) &= \{id\{v\} \mid v \in [\![e]\!]\} \\
[\![\texttt{pkt}]\!](p,\sigma,\phi) &= \{p\} \\
[\![id]\!](p,\sigma,\phi) &= \sigma(id) \\
[\![\texttt{case } \{\cdot\texttt{;default:}e\texttt{;}\}]\!](p,\sigma,\phi) &= [\![e]\!](p,\sigma,\phi) \\
[\![\texttt{case } \{e_1 : e_2\texttt{;cbod;}\}]\!](p,\sigma,\phi) &= \\
\{v_2 \mid (\texttt{true}, v_2) \in [\![(e_1,e_2)]\!](p,\sigma,\phi)\} &\cup \\
\{v_2 \mid \texttt{false} \in [\![e_1]\!](p,\sigma,\phi), v_2 \in [\![\texttt{case } \{\texttt{cbod;}\}]\!]\} & \\
[\![(e_1,\ldots,e_n)]\!](p,\sigma,\phi) &= \{(v_1,\ldots,v_n) \mid v_i \in [\![e_i]\!]\}
\end{aligned}$$

Figure 16: Semantics of expressions

The semantics of `filter` and `assume` only differ when $\{\{(p,\sigma) \mid \texttt{true} \in [\![e]\!](p,\sigma,\phi)\}\} = \{\varnothing\}$. In that case `filter` drops all packets (its semantics is $\{\varnothing\}$), whereas `assume` disallows refinements by denoting $\varnothing$. The semantics of packet field updates (explicit or using `havoc`), conditionals, and let-bindings is standard.

The statement `send` is treated as a function call to the new role we are sending to, putting together all the nondeterministic behaviors of that role with a union. Finally, `fork` makes a cross-product on all the possibilities of each of the statements, generating all possible combinations of multicasts by picking one in each statement of *as*. Composition of statements a, as is defined using a similar cross-product to correctly handle both multicasting and nondeterminism.

A.2.3 Semantics of declarations

The semantics of declarations is given in Figure 18. A declaration updates the environments of functions ϕ and roles ρ. Constraints $\mid e$ and $/ e$ on roles are considered true when unspecified.

A role declaration updates the role environment with a function r. This function first checks whether the conditions e_3 and e_4 are fulfilled (first two lines); then, in the case where this definition is a refinement of an existing role, it checks whether the new role's body is a valid refinement (third line); when those checks pass, r returns the semantics of the body *as* of the role.

A function declaration without an explicit body creates a possible function environment for *any* possible value of this function. When provided a body, those environments are restricted if a prior declaration existed (first line), otherwise an explicit definition is added (second line). Assumptions select the definitions of functions in ϕ that agree with the assumption that is being considered.

$$
\begin{aligned}
[\![a]\!](p,\sigma,\phi,\rho) &\in \mathscr{P}(\mathscr{P}(\texttt{Pkt}\times\texttt{Env})) \\
[\![\texttt{filter}\ e]\!](p,\sigma,\phi,\rho) &= \{\{(p,\sigma) \mid \texttt{true} \in [\![e]\!](p,\sigma,\phi)\}\} && \text{can be } \{\varnothing\} \text{ but not } \varnothing \\
[\![\texttt{assume}\ e]\!](p,\sigma,\phi,\rho) &= \{\{(p,\sigma) \mid \texttt{true} \in [\![e]\!](p,\sigma,\phi)\}\} && \text{only if} \neq \{\varnothing\} \\
[\![\texttt{assume}\ e]\!](p,\sigma,\phi,\rho) &= \varnothing && \text{otherwise} \\
[\![\texttt{pkt}.id := e]\!](p,\sigma,\phi,\rho) &= \{\{(p[id \mapsto v],\sigma)\} \mid v \in [\![e]\!](p,\sigma,\phi)\} \\
[\![\texttt{havoc pkt}.id]\!](p,\sigma,\phi,\rho) &= \{\{(p[id \mapsto v],\sigma)\} \mid v \in \texttt{Val}\} \\
[\![\texttt{if}\ e\ \texttt{then}\ a_1]\!](p,\sigma,\phi,\rho) &= \{(p',\sigma') \in [\![a_1]\!](p,\sigma,\phi,\rho) \mid \texttt{true} \in [\![e]\!](p,\sigma,\phi)\} \cup \\
&\quad \{(p,\sigma) \mid \texttt{false} \in [\![e]\!](p,\sigma,\phi)\} \\
[\![\texttt{if}\ e\ \texttt{then}\ a_1\ \texttt{else}\ a_2]\!](p,\sigma,\phi,\rho) &= \{(p',\sigma') \in [\![a_1]\!](p,\sigma,\phi,\rho) \mid \texttt{true} \in [\![e]\!](p,\sigma,\phi)\} \cup \\
&\quad \{(p',\sigma') \in [\![a_2]\!](p,\sigma,\phi,\rho) \mid \texttt{false} \in [\![e]\!](p,\sigma,\phi)\} \\
[\![\texttt{let}\ id = e]\!](p,\sigma,\phi,\rho) &= \{\{(p,\sigma[id \mapsto v])\} \mid v \in [\![e]\!](p,\sigma,\phi)\} \\
[\![\texttt{send}\ id[e]]\!](p,\sigma,\phi,\rho) &= \bigcup \{\rho(id)(v,p) \mid v \in [\![e]\!](p,\sigma,\phi)\} \\
[\![\texttt{fork}(id \mid e)\ as]\!](p,\sigma,\phi,\rho) &= \bigotimes \{[\![as]\!](p,\sigma[id \mapsto v],\phi,\rho) \mid \texttt{true} \in [\![e]\!](p,\sigma[id \mapsto v],\phi), v \in \texttt{Val}\} \\
\text{where } \bigotimes\{A_1,\ldots,A_n\} &= \{a_1 \cup \ldots \cup a_n \mid a_1 \in A_1,\ldots,a_n \in A_n\}, a_i \in \mathscr{P}(\texttt{Pkt}\times\texttt{Env})
\end{aligned}
$$

$$
\begin{aligned}
[\![as]\!](p,\sigma,\phi,\rho) &\in \mathscr{P}(\mathscr{P}(\texttt{Pkt}\times\texttt{Env})) \\
[\![a,as]\!](p,\sigma,\phi,\rho) &= \bigcup \left\{ \bigotimes \{[\![as]\!](p',\sigma',\phi,\rho) \mid (p',\sigma') \in A\} \mid A \in [\![a]\!](p,\sigma,\phi,\rho) \right\}
\end{aligned}
$$

Figure 17: Semantics of statements

$$
\begin{aligned}
[\![d]\!](\phi,\rho) &\in \Phi \times P \\
[\![\texttt{role}\ id_1\ [id_2]]\!] &= [\![\texttt{role}\ id_1\ [id_2] \mid \texttt{true} \,/\, \texttt{true}]\!] \\
[\![\texttt{role}\ id_1\ [id_2] \mid e]\!] &= [\![\texttt{role}\ id_1\ [id_2] \mid e \,/\, \texttt{true}]\!] \\
[\![\texttt{role}\ id_1\ [id_2] \,/\, e]\!] &= [\![\texttt{role}\ id_1\ [id_2] \mid \texttt{true} \,/\, e]\!] \\
[\![\texttt{role}\ id_1\ [id_2] \mid e_3 \,/\, e_4 = as]\!](\phi,\rho) &= (\phi, \rho\,[id_1 \mapsto r]) \\
\text{where } r = \lambda(p,v_2).\ & \begin{cases} \varnothing \text{ if } \texttt{true} \notin [\![e_3[v_2/id_2]]\!](p,[\,],\phi) \\ \varnothing \text{ if } \texttt{true} \notin [\![e_4[v_2/id_2]]\!](p,[\,],\phi) \\ \varnothing \text{ if } id_1 \in \mathsf{dom}(\rho) \text{ and } [\![as[v_2/id_2]]\!](p,[\,],\phi,\rho) \not\subseteq \rho(id_1) \\ [\![as[v_2/id_2]]\!](p,[\,],\phi,\rho) \text{ otherwise} \end{cases} \\
[\![\texttt{function}\ id_1(id_2)]\!](\phi,\rho) &= (\{\psi[id_1 \mapsto f] \mid \psi \in \phi, f \in (\texttt{Val}\times\texttt{Pkt}\to\texttt{Val}), id_1 \notin \mathsf{dom}(\psi)\}, \rho) \\
[\![\texttt{function}\ id_1(id_2) = e]\!](\phi,\rho) &= (\{\psi \in \phi \mid \forall p,v_2.\psi(id_1)(v_2,p) \in [\![e[v_2/id_2]]\!](p,[\,],\phi), id_1 \in \mathsf{dom}(\psi)\} \cup \\
&\quad \{\psi[id_1 \mapsto f] \mid \psi \in \phi, \forall p,v_2.f(v_2,p) \in [\![e[v_2/id_2]]\!](p,[\,],\phi), id_1 \notin \mathsf{dom}(\psi)\} \\
&\quad ,\rho) \\
[\![\texttt{assume}\ args\ e]\!](\phi,\rho) &= (\{\psi \mid \psi \in \phi, \forall args.\texttt{true} \in [\![e]\!](\{\,\},[args],\{\psi\})\},\rho)
\end{aligned}
$$

$$
[\![d,ds]\!](\phi,\rho) = [\![ds]\!]([\![d]\!](\phi,\rho))
$$

$$
[\![\texttt{refine}\ ids\ ds]\!](\phi,\rho) = [\![ds]\!](\phi,\rho)
$$

$$
[\![r,spec]\!] = [\![spec]\!]([\![r]\!](\phi,\rho))
$$

Figure 18: Semantics of declarations

The semantics of several declarations, refinements and finally whole specifications chains through the semantics of role declarations.

Verifying Reachability in Networks with Mutable Datapaths

Aurojit Panda* Ori Lahav† Katerina Argyraki‡ Mooly Sagiv♢ Scott Shenker*♠
*UC Berkeley †MPI-SWS ‡EPFL ♢TAU ♠ICSI

Abstract

Recent work has made great progress in verifying the forwarding correctness of networks [26–28, 35]. However, these approaches cannot be used to verify networks containing middleboxes, such as caches and firewalls, whose forwarding behavior depends on previously observed traffic. We explore how to verify reachability properties for networks that include such "mutable datapath" elements, both for the original network and in the presence of failures. The main challenge lies in handling large and complicated networks. We achieve scaling by developing and leveraging the concept of slices, which allow network-wide verification to only require analyzing small portions of the network. We show that with slices the time required to verify an invariant on many production networks is independent of the size of the network itself.

1 Introduction

Network operators have long relied on best-guess configurations and a "we'll fix it when it breaks" approach. However, as networking matures as a field, and institutions increasingly expect networks to provide reachability, isolation, and other behavioral invariants, there is growing interest in developing rigorous verification tools that can check whether these invariants are enforced by the network configuration.

The first generation of such tools [26–28, 35] check reachability and isolation invariants in near-real time, but assume that network devices have "static datapaths, " *i.e.,* their forwarding behavior is set by the control plane and not altered by observed traffic. This assumption is entirely sufficient for networks of routers but not for networks that contain middleboxes with "mutable datapaths" whose forwarding behavior may depend on the entire packet history they have seen. Examples of such middleboxes include firewalls that allow end hosts to establish flows to the outside world through "hole punching" and network optimizers that cache popular content. Middleboxes are prevalent – in fact, they constitute as much as a third of network devices in enterprise networks [49] – and expected to become more so with the rise of Network Function Virtualization (NFV) because the latter makes it easy to deploy additional middleboxes without changes in the physical infrastructure [13]. Given their complexity and prevalence, middleboxes are the cause of many network failures; for instance, 43% of a network provider's failure incidents involved middleboxes, and between 4% and 15% of these incidents were the result of middlebox misconfiguration [41].

Our goal is to reduce such misconfigurations by extending verification to large networks that contain middleboxes with mutable datapaths. In building our system for verifying reachability and isolation properties in mutable networks – which we call VMN (for *v*erifying *m*utable *n*etworks) – we do not take the direct approach of attempting to verify middlebox code itself, and then extend this verification to the network as a whole, for two reasons. First, such an approach does not scale to large networks. The state-of-the-art in verification is still far from able to automatically analyze the source code or executable of most middleboxes, let alone the hundreds of interconnected devices that it interacts with [51]. Thus, verifying middlebox code directly is *practically infeasible*.

Second, middlebox code does not always work with easily verified abstractions. For example, some IDSes attempt to identify suspicious traffic. No method can possibly verify whether their code is successful in identifying all suspicious traffic because there is no accepted definition of what constitutes suspicious. Thus, verifying such middlebox code is *conceptually impossible*.

Faced with these difficulties, we return to the problem operators want to solve. They recognize that there may be imprecision in identifying suspicious traffic, but they want to ensure that all traffic that the middlebox identifies as being suspicious is handled appropriately (*e.g.,* by being routed to a scrubber for further analysis). The first problem – perfectly identifying suspicious traffic – is not only ill-defined, it is not controlled by the operator (in the sense that any errors in identification are beyond the reach of the operator's control). The second problem – properly handling traffic considered suspicious by a middlebox – is precisely what an operator's configuration, or misconfiguration, can impact.

The question, then is how to abstract away unnecessary complexity so that we can provide useful answers to operators. We do so by leveraging two insights. First,

middlebox functionality can be logically divided into two parts: forwarding (*e.g.,* forward suspicious and non-suspicious packets through different output ports) and packet classification (*e.g.,* whether a packet is suspicious or not). Verifying this kind of amorphous packet classification it not our concern. Second, there exist a large number of different middleboxes, but most of them belong to a relatively small set of middlebox types – firewalls, IDS/IPSes, and network optimizers (the latter police traffic, eliminate traffic redundancy, and cache popular content) [6]; middleboxes of the same type define similar packet classes (*e.g.,* "suspicious traffic") and use similar forwarding algorithms, but may differ dramatically in how they implement packet classification.

Hence, we model a middlebox as: a *forwarding model*, a set of *abstract packet classes*, and a set of *oracles* that automatically determine whether a packet belongs to an abstract class – so, the oracle abstracts away the implementation of packet classification. With this approach, we do not need a new model for every middlebox, only one per middlebox type.

This modeling approach avoids the conceptual difficulties, but does not address the practical one of scaling to large networks. One might argue that, once we abstract away packet classification, what remains of middlebox functionality is simple forwarding logic (how to forward each packet class), hence it should be straightforward to extend prior tools to handle middleboxes. However, while checking reachability property in static networks is PSPACE-complete [2], it is EXPSPACE-complete when mutable datapaths are considered [56]. Mutable verification is thus algorithmically more complicated. Furthermore, recent work has shown that even verifying such properties in large static networks requires the use of "reduction techniques", which allow invariants to be verified while reasoning about a small part of the network [40]. Applying such techniques to mutable datapaths is more complex, because parts of the network may *effect* each other through state, without explicitly exchanging traffic – making it hard to partition the network.

To address this, we exploit the fact that, even in networks with mutable datapaths, observed traffic often affects only a well-defined subset of future traffic, *e.g.,* packets from the same TCP connection or between the same source and destination. We formalize this behavior in the form of two middlebox properties: *flow-parallelism* and *origin-independence*; when combined with structural and policy symmetry, as is often the case in modern networks [40], these properties enable us to use reduction effectively and verify invariants in arbitrarily large networks in a few seconds (§5).

The price we pay for model simplicity and scalability is that we cannot use our work to check middlebox implementations and catch interesting middlebox-specific bugs [10]; however, we argue that it makes sense to develop separate tools for that purpose, and not unnecessarily complicate verification of reachability and isolation.

Listing 1: Model for an example firewall

```
1  class Firewall (acls:
       ↪ Set[(Address, Address)]) {
2    abstract malicious(p: Packet): bool
3    val tainted: Set[Address]
4    def model (p: Packet) = {
5      tainted.contains(p.src)
           ↪ => forward(Empty)
6      acls.contains((p.src,
           ↪ p.dst)) => forward(Empty)
7      malicious(p)
           ↪ => tainted.add(p.src);
           ↪ forward(Empty)
8      _ => forward(Seq(p))
9    }
10 }
```

2 Modeling Middleboxes

We begin by looking at how middleboxes are modeled in VMN. First, we provide a brief overview of how these models are expressed (§2.1), then we present the rationale behind our choices (§2.2), and finally we discuss discuss real-world examples (§2.3).

2.1 Middlebox Models

We illustrate our middlebox models through the example in Listing 1, which shows the model for a simplified firewall. The particular syntax is not important to our technique; we use a Scala-like language, because we found the syntax to be intuitive for our purpose, and in order to leverage the available libraries for parsing Scala code. We limit our modeling language to not support looping (*e.g.,* `for`, `while`, etc.) and only support branching through partial functions (Lines 5–7).

A VMN model is a class that implements a `model` method (Line 4). It may define *oracles* (*e.g.,* `malicious` on Line 3), which are abstract functions with specified input (`Packet` in this case) and output (`boolean` in this case) type. It may also define data structures—sets, lists, and maps—to store state (*e.g.,* `tainted` on Line 3) or to accept input that specifies configuration (*e.g.,* `acls` on Line 1). Finally, it may access predefined packet-header fields (*e.g.,* `p.src` and `p.dst` on Lines 6 and 7). We limit function calls to oracles and a few built-in functions for extracting information from packet-header fields (our example model does not use the latter).

The `model` method specifies the middlebox's forwarding behavior. It consists of a set of variable declarations, followed by a set of guarded forwarding rules (Lines 5–8). Each rule must be terminated by calling the `forward`

function with a set of packets to forward (which could be the empty set). In our example, the first three rules (Line 5–7) drop packets by calling `forward` with an empty set, while the last one (Line 8) forwards the received packet `p`.

Putting it all together, the model in Listing 1 captures the following middlebox behavior: On receiving a new packet, first check if the packet's source has previously contributed malicious packets, in which case drop the packet (Line 5). Otherwise, check if the packet is prohibited by the provided access control list (ACL), in which case drop the packet (Line 6). Otherwise, checks if the packet is malicious, in which case record the packet's source and drop the packet (Line 7). Finally, if none of these conditions are triggered, forwards the packet (Line 8).

The model in Listing 1 does *not* capture how the firewall determines whether a packet is malicious or not; that part of its functionality is abstracted away through the `malicious` oracle. We determine what classification choices are made by an oracle (*e.g.,* `malicious`) and what are made as a part of the forwarding model (*e.g.,* our handling of ACLs) based on whether the packet can be fully classified by just comparing header fields to known values (these values might have been set as a part of processing previous packets) – as is the case with checking ACLs and whether a flow is tainted – or does it require more complex logic (*e.g.,* checking the content, etc.) – as is required to mark a packet as malicious.

2.2 Rationale and Implications

Why did we choose to model middleboxes as a *forwarding model* which can call a set of *oracles*?

First, we wanted to express middlebox behavior in the same terms as those used by network operators to express reachability and isolation invariants. Network operators typically refer to traffic they wish to allow or deny in two different ways: in terms of packet-header fields that have semantic meaning in their network (*e.g.,* a packet's source IP address indicates the particular end host or user that generated that packet), or in terms of semantic labels attached to packets or flows by middleboxes (*e.g.,* "contains exploits," "benign," or "malicious"). This is why a VMN model operates based on two kinds of information for each incoming packet: predefined header fields and abstract packet classes defined by the model itself, which represent semantic labels.

Second, like any modeling work, we wanted to strike a balance between capturing relevant behavior and abstracting away complexity. Two elements guided us: first, the middlebox configuration that determines which semantic labels are attached to each packet is typically not written by network operators: it is either embedded in middlebox code, or provided by third parties, *e.g.,* Emerging Threats rule-set [12] or vendor provided virus definitions [52]. Second, the middlebox code that uses such rulesets and/or virus definitions is typically sophisticated and performance-optimized, *e.g.,* IDSes and IPSes typically extract the packet payload, reconstruct the byte stream, and then use regular expression engines to perform pattern matches. So, the part of middlebox functionality that maps bit patterns to semantic labels (*e.g.,* determines whether a packet sequence is "malicious") is hard to analyze, yet unlikely to be of interest to an operator who wants to check whether they configured their network as intended. This is why we chose to abstract away this part with the oracles – and model each middlebox only in terms of how it treats a packet based on the packet's headers and abstract classes.

Mapping low-level packet-processing code to semantic labels is a challenge that is common to network-verification tools that handle middleboxes. We address it by explicitly abstracting such code away behind the oracles. Buzz [14] provides ways to automatically derive models from middlebox code, yet expects the code to be written in terms of meaningful semantics like addresses. In practice, this means that performance-optimized middleboxes (*e.g.,* ones that build on DPDK [22] and rely on low level bit fiddling) need to be hand-modeled for use with BUZZ. Similarly, SymNet [51] claims to closely matches executable middlebox code. However, SymNet also requires that code be written in terms of access to semantic fields; in addition, it allows only limited use of state and limits loops. In reality, therefore, neither Buzz nor SymNet can model the behavior of general middleboxes (*e.g.,* IDSes and IPSes). We recognize that modeling complex classification behavior, *e.g.,* from IDSes and IPSes, either by expressing these in a modeling language or deriving them from code is impractical, and of limited practical use when verifying reachability and isolation. Therefore rather than ignoring IDSes and IPSes (as done explicitly by SymNet, and implicitly by Buzz), we use oracles to abstract away classification behavior.

How many different models? We need one model per middlebox type, *i.e.,* one model for all middleboxes that define the same abstract packet classes and use the same forwarding algorithm. A 2012 study showed that, in enterprise networks, most middleboxes belong to a small number of types: firewalls, IDS/IPS, and network optimizers [49]. As long as this trend continues, we will also need a small number of models.

Who will write the models? Because we need only a few models, and they are relatively simple, they can come from many sources. Operators might provide them as part of their requests for bids, developers of network-configuration checkers (*e.g.,* Veriflow Inc. [57] and Forward Network [15]) might develop them as part of their offering, and middlebox manufactures might provide them to enable reliable configuration of networks which deploy their products. The key point is that one can write a VMN model without access to the corresponding middlebox's source code or executable; all one needs is

the middlebox's manual—or any document that describes the high-level classification performed by the middlebox defines, whether and how it modifies the packets of a given class, and whether it drops or forwards them.

What happens to the models as middleboxes evolve? There are two ways in which middleboxes evolve: First, packet-classification algorithms change, *e.g.,* what constitutes "malicious" traffic evolves over time. This does not require any update to VMN models, as they abstract away packet-classification algorithms. Second semantic labels might change (albeit more slowly), *e.g.,* an operator may label traffic sent by a new applications. A new semantic label requires updating a model with a new oracle and a new guided forwarding rule.

Limitations: Our approach cannot help find bugs in middlebox code—*e.g.,* in regular expression matching or flow lookup—that would cause a middlebox to forward packets it should be dropping or vice versa. In our opinion, it does not make sense to incorporate such functionality in our tool: such debugging is tremendously simplified with access to middlebox code, while it does not require access to the network topology where the middlebox will be placed. Hence, we think it should be targeted by separate debugging tools, to be used by middlebox developers, not network operators trying to figure out whether they configured their network as they intended.

2.3 Real-world Examples

Next, we present some examples of how existing middleboxes can be modeled in VMN. For brevity, we show models for firewalls and intrusion prevention systems in this section. We include other examples including NATs (from `iptables` and pfSense), gateways, load-balancers (HAProxy and Maglev [11]) and caches (Squid, Apache Web Proxy) in Appendix A. We also use Varnish [20], a protocol accelerator to demonstrate VMN's limitations.

Firewalls We examined two popular open-source firewalls, `iptables` [42] and pfSense [38], written by different developers and for different operating systems (`iptables` targets Linux, while pfSense requires FreeBSD). These tools also provide NAT functions, but for the moment we concentrate on their firewall functions.

For both firewalls, the configuration consists of a list of match-action rules. Each action dictates whether a matched packet should be forwarded or dropped. The firewall attempts to match these rules in order, and packets are processed according to the first rule they match. Matching is done based on the following criteria:

- Source and destination IP prefixes.
- Source and destination port ranges.
- The network interface on which the packet is received and will be sent.
- Whether the packet belongs to an established connection, or is "related" to an established connection.

The two firewalls differ in how they identify related connections: `iptables` relies on independent, protocol-specific helper modules [32]. pfSense relies on a variety of mechanisms: related FTP connections are tracked through a helper module, related SIP connections are expected to use specific ports, while other protocols are expected to use a pfSense proxy and connections from the same proxy are treated as being related.

Listing 2 shows a VMN model that captures the forwarding behavior of both firewalls—and, to the best of our knowledge, any shipping IP firewall. The configuration input is a list of rules (Line 12). The `related` oracle abstracts away the mechanism for tracking related connections (Line 13). The `established` set tracks established connections (Line 14). The forwarding model searches for the first rule that matches each incoming packet (Lines 17–26); if one is found and it allows the packet, then the packet is forwarded (Line 27), otherwise it is dropped (Line 28).

Listing 2: Model for `iptables` and pfSense

```
1  case class Rule (
2    src: Option[(Address, Address)],
3    dst: Option[(Address, Address)],
4    src_port: Option[(Int, Int)],
5    dst_port: Option[(Int, Int)],
6    in_iface: Option[Int],
7    out_iface: Option[Int],
8    conn: Option[Bool],
9    related: Option[Bool],
10   accept: Bool
11 )
12 class Firewall (acls: List[Rule]) {
13   abstract related (p: Packet): bool
14   val established: Set[Flow]
15   def model (p: Packet) = {
16     val f = flow(p);
17     val match = acls.findFirst(
18       acl => (acl.src.isEmpty ||
19               acl.src._1 <=
                    ↪ p.src && p.src
                    ↪ < acl.src._2) &&
20              ...
21              (acl.conn.isEmpty ||
22               acl.conn ==
                    ↪ established(f))&&
23              (acl.related.isEmpty ||
24               acl.related
                    ↪ == related(f)));
25     match.isDefined && match.accept
         ↪ => forward(Seq(p))
         ↪ // We found
         ↪ a match which said the
         ↪ packet should be forwarded
26     _ => forward(Empty)
         ↪ // Drop all other packets
27   }
28 }
```

Intrusion Prevention Systems We considered two kinds: general systems like Snort [45], which detect a variety of attacks by performing signature matching on packet payloads, and web application firewalls like ModSecurity [44] and IronBee [43], which detect attacks on web applications by analyzing HTTP requests and bodies sent to a web server. Both kinds accept as configuration "rulesets" that specify suspicious payload strings and other conditions, *e.g.,* TCP flags, connection status, etc., that indicate malicious traffic. Network operators typically rely on community maintained rulesets, *e.g.,* Emerging Threats [12] (with approximately 20,000 rules) and Snort rulesets (with approximately 3500 rules) for Snort; and OWASP [24] for ModSecurity and IronBee.

Listing 3: Model for IPS

```
1 class IPS {
2   abstract malicious(p: Packet): bool
3   val infected: Set[Flow]
4   def model (p: Packet) = {
5     infected.contains(flow(p))
        ↪ => forward(Empty)
6     malicious(p)
        ↪ => infected.add(flow(p);
        ↪ forward(Empty)
7     _ => forward(Seq(p))
8   }
9 }
```

Listing 3 shows a VMN model that captures the forwarding behavior of these systems. The `malicious` oracle abstracts away how malicious packets are identified (Line 2). The `infected` set keeps track of connections that have contributed malicious packets (Line 3). The forwarding model simply drops a packet if the connection is marked as infected (Line 5) or is malicious according to the oracle (Line 6). It may seem counter-intuitive that this model is simpler than that of a firewall (Listing 2), because we tend to think of IDS/IPSes as more complex devices; however, a firewall has more sophisticated forwarding behavior, which is what we model, whereas the complexity of an IDS/IPS lies in packet classification, which is what we abstract away.

Programmable web accelerators The Varnish cache [20] demonstrates the limits of our approach. Varnish allows the network operator to specify, in detail, the handling of each HTTP(S) request, *e.g.,* building dynamic pages that reference static cached content, or even generating simple dynamic pages. Varnish's configuration therefore acts more like a full-fledged program – and it is hard to separate out "configuration" from forwarding implementation. We can model Varnish by either developing a model for each configuration or abstracting away the entire configuration. The former impedes reuse, while the later is imprecise and neither is suitable for verification.

3 Modeling Networks

Having described VMN's middlebox models, we turn to how VMN models an entire network of middleboxes and how we scale verification to large networks. Here we build on existing work on static network verification [27,28]. For scalability, which is one of our key contributions, we identify small network subsets—*slices*—where we can check invariants efficiently. For some common middleboxes we can find slices whose size is independent of network size.

3.1 Network Models

Veriflow [28] and header-space analysis [27] (HSA) summarize network behavior as a *network transfer function*. This builds on the concept of a *located packet*, *i.e.,* a packet augmented with the input port on which it is received. A network transfer function takes as input a located packet and produces a set of located packets. The transfer function ensures that output packets are located at feasible ports, *i.e.,* at ports physically connected to the input location.

VMN models a network as a collection of end hosts and middleboxes connected by a network transfer function. More formally, we define a network $N = (V, E, P)$ to be a set of nodes V, a set of links (edges) E connecting the nodes, and a possibly infinite set of packets P. Nodes include both end hosts and middleboxes. For notational convenience, each packet $p \in P$ includes its network location (port), which is given by $p.loc$. For such a network N, we define the network transfer function N_T as

$$N_T : p \rightarrow P' \subseteq P,$$

where $p \in P$ is a packet and $P' \subseteq P$ is a set of packets.

Given a network, we treat all end hosts and middleboxes as endpoints from which packets can be sent and at which they can be received; we then use Veriflow to generate a transfer function for this network, and we turn the resulting output into a form that can be accepted by an SMT solver. This essentially transforms the network into one where all packets sent from end hosts traverse a sequence of middleboxes before being delivered to their final destination. Our verification then just focuses on checking whether the sequence of middleboxes encountered by a packet correctly enforces any desired reachability invariant. Note that, if a reachability invariant is enforced merely by static-datapath—e.g., router—configuration, we do successfully detect that, i.e., VMN's verification is a generalization of static network verification.

Veriflow and HSA cannot verify networks with forwarding loops, and we inherit this limitation; we check to ensure that the network does not have any forwarding loop and raise an error whenever one is found.

3.2 Scaling Verification: Slicing

While network transfer functions reduce the number of distinct network elements that need to be considered

in verification, this reduction is often insufficient. For example, Microsoft's Chicago Datacenter [8] contains over 224,000 servers running virtual machines connected over virtual networks. In such an environment, each server typically acts as a middlebox, resulting in a network with several 100,000 middleboxes, *e.g.,* firewalls, load balancers, SSL proxies, etc.. We want to be able to check invariants in such large networks within a few minutes, however, typically verifying such large instances is infeasible or takes several days.

Our approach to achieving this goal is to identify subnetworks which can be used to efficiently verify invariants. We provide a brief overview of our techniques in this section, and deal a more complete treatment to Appendix C.

First, we formally define a *subnetwork*: Given a network $N = (V, E, P)$ with network transfer function N_T, a subnetwork Ω of N is a subgraph of N consisting of: a subset of the nodes in V; all links in E that connect this subset of nodes; and all packets in P whose location, source and destination *are in* Ω. We say that a packet's source (destination) *is in* Ω, if and only if a node in Ω has the right to use the packet' source (destination) address.[1] We compute a restricted transfer function $N_T|_\Omega$ for subnetwork Ω by modifying N_T such that its domain and range are restricted to packets in Ω. We say that subnetwork Ω is *closed under forwarding*, if, for any packet p in Ω, $N_T|_\Omega(p) = N_T(p)$, *i.e.,* the packet is not forwarded out of Ω.

A *slice* is a special kind of subnetwork. We treat the network as a state machine whose state is defined by the set of packet that have been delivered, the set of packets pending delivery and the state of all middleboxes (see Appendix C for details). State transitions in this model represent the creation of a new packet at an endhost or the delivery and processing of a pending packet at a node. We say that a state S is *reachable in the network*, if and only if there is a valid sequence of transitions starting from an initial state[2] that results in the network being in state S. A subnetwork Ω is *closed under state*, if and only if (a) it is closed under forwarding and (b) every state that is reachable in the entire network has an equivalent reachable state in Ω (*i.e.,* a surjection exists between the network state and subnetwork's state). A slice is a subnetwork that is closed under state.

In our formalism, an invariant I is a predicate on the state of a network, and an invariant is violated if and only if the predicate does not hold for a reachable state. We say an invariant is *evaluable on a subnetwork* Ω, if the corresponding predicate refers only to packets and middlebox state contained within Ω. As we show in Appendix C, any invariant evaluable on some slice Ω of network N, holds in Ω if and only if it also holds in N.

[1] We assume that we are provided with a mapping from each node to the set of addresses that it can use.

[2] The initial state represents a network where no packets have been created or delivered.

The remaining challenge is to identify such a slice given a network N and an invariant I, and we do so by taking advantage of how middleboxes update and use state. We identify two types of middleboxes: *flow-parallel* middlebox, whose state is partitioned such that two distinct flows cannot affect each other; and *origin-agnostic* middleboxes whose behavior is not affected by the origin (*i.e.,* sequence of transitions) of its current state. If all middleboxes in the network have one of these properties, then invariants can be verified on slices whose size is independent of the size of the network.

3.2.1 Flow-Parallel Middleboxes

Several common middleboxes partition their state by flows (*e.g.,* TCP connections), such that the handling of a packet is dictated entirely by its flow and is independent of any other flows in the network. Examples of such middleboxes include firewalls, NATs, IPSes, and load balancers. For analysis, such middleboxes can be decomposed into a set of middleboxes, each of which processes exactly one flow without affect network behavior. From this observation we deduce that any subnetwork that is closed under forwarding and contains only flow-parallel middleboxes is also closed under state, and is therefore a slice.

Therefore, , if a network N contains only flow-parallel middleboxes, then we can find a slice on which invariant I is evaluable by picking the smallest subnetwork Ω on which I is evaluable (which always exists) and adding the minimal set of nodes from network N required to ensure that it is closed under forwarding. This minimal set of nodes is precisely the set of all middleboxes that appear on any path connecting hosts in Ω. Since path lengths in a network are typically independent of the size of the network, the size of a slice is generally independent of the size of the network. We present an example using slices comprised of flow-parallel middleboxes, and evaluate its efficiency in §5.1.

3.2.2 Origin Agnostic Middleboxes

Even when middleboxes, *e.g.,* caches, must share state across flows, their behavior is often dependent only on the state of a middlebox not on the sequence of transitions leading up to that state, we call such middleboxes *origin-agnostic*. Examples of origin-agnostic middleboxes include caches—whose behavior depends only on whether some content is cached or not; IDSes, etc. We also observe that network policy commonly partitions end hosts into *policy equivalence classes*, *i.e.,* into set of end hosts to which the same policy applies and whose packets are treated equivalently. In this case, any subnetwork that is closed under forwarding, and contains only origin-agnostic (or flow-parallel) middleboxes, and has an end host from each policy equivalence class in the network is also closed under state, hence, it is a slice.

Therefore, given a network N containing only flow-

parallel and origin-agnostic middleboxes and an invariant I, we can find a slice on which I is evaluable by using a procedure similar to the one for flow-parallel middleboxes. This time round, the slice must contain an end host from each policy equivalence class, and hence the size of the slice depends on the number of such classes in the network. Networks with complex policy are harder to administer, and generally scaling a network does not necessitate adding more policy classes. Therefore, the size of slices in this case is also independent of the size of the network. We present an example using slices comprised of origin-agnostic middleboxes, and evaluate its efficiency in §5.2.

3.3 Scaling Verification: Symmetry

Slicing allows us to verify an individual invariant in an arbitrarily large network. However, the correctness of an entire network depends on several invariants holding simultaneously, and the number of invariants needed to prove that an entire network is correct grows with the size of the network. Therefore, to scale verification to the entire network we must reduce the number of invariants that need to be checked. For this we turn to symmetry; we observe that networks (especially as modeled in VMN) have a large degree of symmetry, with traffic between several end host pairs traversing the same sequence of middlebox types, with identical configurations. Based on this observation, and on our observation about *policy equivalence classes* we can identify invariants which are symmetric; we define two invariants I_1 and I_2 to be symmetric if I_1 holds in the network N if and only if I_2 also holds in the network. Symmetry can be determined by comparing the smallest slices in which each invariant can be evaluated, and seeing if the graphs are isomorphic up to the type and configuration of individual middleboxes. When possible VMN uses symmetry to reduce the number of invariants that need to be verified, enabling correctness to be checked for the entire network.

4 Checking Reachability

VMN accepts as input a set of middlebox models connected through a network transfer function representing a *slice* of the original network, and one or more invariants; it then runs a *decision procedure* to check whether the invariants hold in the given slice or are *violated*. In this section we first describe the set of invariants supported by VMN (§4.1) and then describe the decision procedure used by VMN (§4.2).

4.1 Invariants

VMN focuses on checking *isolation invariants*, which are of the form "do *packets* (or *data*) belonging to some *class* reach one or more *end hosts*, given certain *conditions* hold?" We say that an invariant holds if and only if the answer to this question is no. We verify this fact assuming worse-case behavior from the oracles. We allow invariants to *classify* packet or data using (a) header fields (source address, destination address, ports, etc.); (b) origin, indicating what end host originally generated some content; (c) oracles *e.g.,* based on whether a packet is *infected*, etc.; or (d) any combination of these factors, *i.e.,* we can choose packets that belong to the intersection of several classes (using logical conjunction), or to the union of several classes (using disjunction). Invariants can also specify temporal *conditions* for when they should hold, *e.g.,* invariants in VMN can require that packets (in some class) are blocked until a particular packet is sent. We now look at a few common classes of invariants in VMN.

Simple Isolation Invariants are of the form "do packets belonging to some class reach destination node d?" This is the class of invariants supported by stateless verification tools *e.g.,* Veriflow and HSA. Concrete examples include "Do packets with source address a reach destination b?", "Do packets sent by end host a reach destination b?", "Are packets deemed mallicious by Snort delivered to server s?", etc.

Flow Isolation Invariants extend simple isolation invariants with temporal constraints. This includes invariants like "Can a receive a packet from b without previously opening a connection?"

Data Isolation Invariants make statements about what nodes in the network have access to some data, this is similar to invariants commonly found in information flow control [59] systems. Examples include "Can data originating at server s be accessed by end host b?'

Pipeline Invariants ensure that packets of a certain class have passed through a particular middlebox. Examples include "Do all packets marked as suspicious by a light IDS pass through a scrubber before being delivered to end host b?"

4.2 Decision Procedure

VMN's verification procedure builds on Z3 [9], a modern SMT solver. Z3 accepts as input logical formulae (written in first-order logic with additional "theories" that can be used to express functions, integers and other objects) which are expressed in terms of variables, and returns whether there exists a satisfying assignment of values to variables, *i.e.,* whether their exists an assignment of values to variables that render all of the formulae true. Z3 returns an example of a satisfying assignment when the input formulae are *satisfiable*, and labels them *unsatisfiable* otherwise. Checking the satisfiability of first-order logic formuals is undecidable in general, and even determining whether satisfiablity can be successfully checked for a particular input is undecidable. As a result Z3 relies on timeouts to ensure termination, and reports *unknown* when a timeout is triggered while checking satifiability.

The input to VMN's verification procedure is comprised of a set of invariants and a network slice. The network slice is expressed as a set of middlebox models, a set of middle-

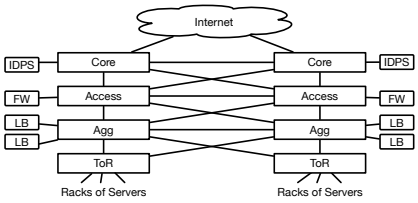


Figure 1: Topology for a datacenter network with middleboxes from [41]. The topology contains firewalls (**FW**), load balancers (**LB**) and intrusion detection and prevention systems (**IDPS**).

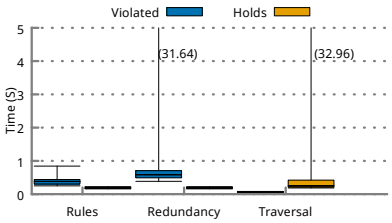


Figure 2: Time taken to verify each network invariant for scenarios in §5.1. We show time for checking both when invariants are violated (Violated) and verified (Holds).

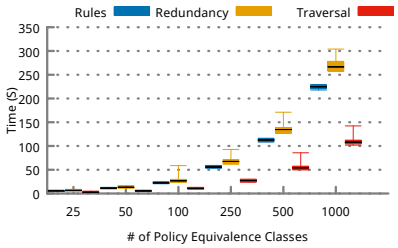


Figure 3: Time taken to verify all network invariants as a function of policy complexity for §5.1. The plot presents minimum, maximum, 5^{th}, 50^{th} and $95th$ percentile time for each.

box instances, a set of end hosts and a network transfer function connecting these nodes. VMN converts this input into a set of first order formulae, which we refer to as the *slice model*. We give details of our logical models in Appendix B, but at a high-level: VMN first adds formulas for each middlebox instance, this formula is based on the provided models; next it compiles the network transfer function into a logical formula; finally it adds formula for each end host.

In VMN, invariants are expressed as logical formula, and we provide convenience functions designed to simplify the task of writing such invariants. As is standard in verification the logical formula representing an invariant is a negation of the invariant itself, and as a result the logical formulation is *satisfiable* if and only if the invariants is *violated*.

VMN checks invariants by invoking Z3 to check whether the conjunction of the slice model and invariants is satisfiable. We report that the invariant holds if and only if Z3 proves this formal to be unsatisfiable, and report the invariant is vilated when a satisfying assignment is found. A convenient feature of such a mechanism is that when a violation is found we can use the satisfying assignment to generate an example where the invariant is violated. This is useful for diagnosing and fixing the configuration error that led to the violation.

Finally, as shown in Appendix D, the formulae generated by VMN lie in a fragment of first order logic called EPR-F, which is an extension of "Effectively Propositional Reasoning" (EPR) [39] a decidable fragment of first-order logic. The logical models produced by VMN are therefore decidable, however when verifying invariants on large network slices Z3 might timeout, and thus VMN may not always be able to determine whether an invariant holds or is violated in a network. In our experience, verification of invariants generally succeeds for slices with up to a few hundred middleboxes.

5 Evaluation

To evaluate VMN we first examine how it would deal with several real-world scenarios and then investigate how it scales to large networks. We ran our evaluation on servers running 10-core, 2.6GHz Intel Xeon processors with 256 GB of RAM. We report times taken when verification is performed using a single core. Verification can be trivially parallelized over multiple invariants. We used Z3 version 4.4.2 for our evaluation. SMT solvers rely on randomized search algorithms, and their performance can vary widely across runs. The results reported here are generated from 100 runs of each experiment.

5.1 Real-World Evaluation

A previous measurement study [41] looked at more than 10 datacenters over a 2 year period, and found that configuration bugs (in both middleboxes and networks) are a frequent cause of failure. Furthermore, the study analyzed the use of redundant middleboxes for fault tolerance, and found that redundancy failed due to misconfiguration roughly 33% of the time. Here we show how VMN can detect and prevent the three most common classes of configuration errors, including errors affecting fault tolerance. For our evaluation we use a datacenter topology (Figure 1) containing 1000 end hosts and three types of middleboxes: stateful firewalls, load balancers and intrusion detection and prevention systems (IDPSs). We use redundant instances of these middleboxes for fault tolerance. We use load balancers to model the effect of faults (*i.e.,* the load balancer can non-deterministically choose to send traffic to a redundant middlebox). For each scenario we report time taken to verify a single invariant (Figure 2), and time taken to verify all invariants (Figure 3); and show how these times grow as a function of policy complexity (as measured by the number of policy equivalence classes). Each box and whisker plot shows minimum, 5^{th} percentile, median, 95^{th} percentile and maximum time for verification.

Incorrect Firewall Rules: According to [41], 70% of all reported middlebox misconfiguration are attributed to incorrect rules installed in firewalls. To evaluate this scenario we begin by assigning each end host to one of a few policy groups.[3] We then add firewall rules to prevent end hosts in one group from communicating with end hosts in any other group. We introduce misconfiguration by deleting a random set of these firewall rules. We use VMN to identify for which end hosts the desired invariant

[3]Note, policy groups are distinct from policy equivalence class; a policy group signifies how a network administrator might group end hosts while configuring the network, however policy equivalence classes are assigned based on the actual network configuration.

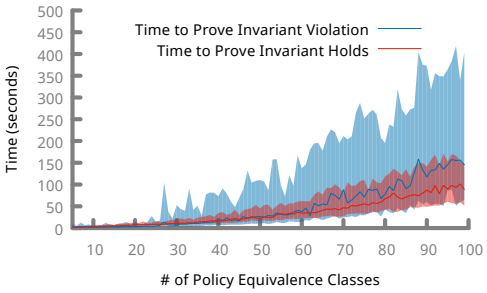


Figure 4: Time taken to verify each data isolation invariant. The shaded region represents the 5^{th}–95^{th} percentile time.

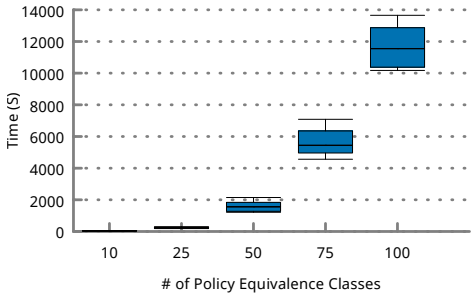


Figure 5: Time taken to verify all data isolation invariants in the network described in §5.2.

holds (*i.e.,* that end hosts can only communicate with other end hosts in the same group). Note that all middleboxes in this evaluation are flow-parallel, and hence the size of a slice on which invariants are verified is independent of both policy complexity and network size. In our evaluation, we found that VMN correctly identified all violations, and did not report any false positives. The time to verify a single invariant is shown in Figure 2 under Rules. When verifying the entire network, we only need to verify as many invariants as policy equivalence classes; end hosts affected by misconfigured firewall rules fall in their own policy equivalence class, since removal of rules breaks symmetry. Figure 3 (Rules) shows how whole network verification time scales as a function of policy complexity.

Misconfigured Redundant Firewalls Redundant firewalls are often misconfigured so that they do not provide fault tolerance. To show that VMN can detect such errors we took the networks used in the preceding simulations (in their properly configured state) and introduced misconfiguration by removing rules from some of the backup firewall. In this case invariant violation would only occur when middleboxes fail. We found VMN correctly identified all such violations, and we show the time taken for each invariant in Figure 2 under "Redundant", and time taken for the whole network in Figure 3.

Misconfigured Redundant Routing Another way that redundancy can be rendered ineffective by misconfiguration is if routing (after failures) allows packets to bypass the middleboxes specified in the pipeline invariants. To test this we considered, for the network described above, an invariant requiring that all packet in the network traverse an IDPS before being delivered to the destination end host. We changed a randomly selected set of routing rules so that some packets would be routed around the redundant IDPS when the primary had failed. VMN correctly identified all such violations, and we show times for individual and overall network verification under "Traversal" in Figures 2 and 3.

We can thus see that verification, as provided by VMN, can be used to prevent many of the configuration bugs reported to affect today's production datacenters. Moreover, the verification time scales linearly with the number of policy equivalence classes (with a slope of about three invariants per second). We now turn to more complicated invariants involving data isolation.

5.2 Data Isolation

Modern data centers also run storage services such as S3 [46], AWS Glacier [19], and Azure Blob Store [3]. These storage services must comply with legal and customer requirements [37] limiting access to this data. Operators often add caches to these services to improve performance and reduce the load on the storage servers themselves, but if these caches are misplaced or misconfigured then the access policies could be violated. VMN can verify these data isolation invariants.

To evaluate this functionality, we used the topology (and correct configuration) from §5.1 and added a few content caches by connecting them to top of rack switches. We also assume that each policy group contains separate servers with private data (only accessible within the policy group), and servers with public data (accessible by everyone). We then consider a scenario where a network administrator inserts caches to reduce load on these data servers. The content cache is configured with ACL entries[4] that can implement this invariant. Similar to the case above, we introduce configuration errors by deleting a random set of ACLs from the content cache and firewalls.

We use VMN to verify data isolation invariants in this network (*i.e.,* ensure that private data is only accessible from within the same policy group, and public data is accessible from everywhere). VMN correctly detects invariant violations, and does not report any false positives. Content caches are origin agnostic, and hence the size of a slice used to verify these invariants depends on policy complexity. Figure 4 shows how time taken for verifying each invariant varies with the number of policy equivalence classes. In a network with 100 different policy equivalence classes, verification takes less than 4 minutes on average. Also observe that the variance for verifying a single invari-

[4]This is a common feature supported by most open source and commercial caches.

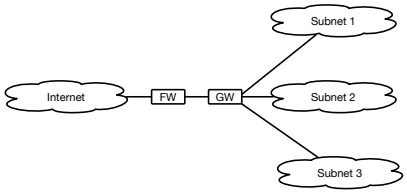


Figure 6: Topology for enterprise network used in §5.3.1, containing a firewall (**FW**) and a gateway (**GW**).

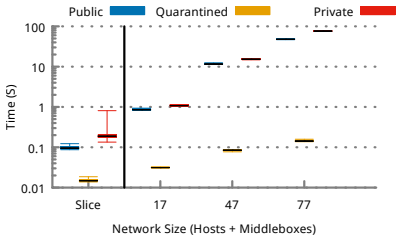


Figure 7: Distribution of verification time for each invariant in an enterprise network (§5.3.1) with network size. The left of the vertical line shows time taken to verify a slice, which is independent of network size, the right shows time taken when slices are not used.

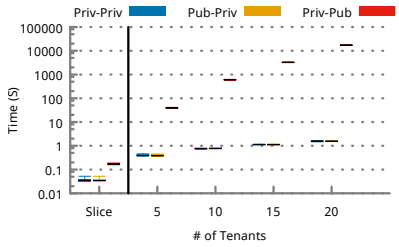


Figure 8: Average verification time for each invariant in a multi-tenant datacenter (§5.3.2) as a function of number of tenants. Each tenant has 10 end hosts. The left of the vertical line shows time taken to verify a slice, which is independent of the number of tenants.

ant grows with the size of slices used. This shows one of the reasons why the ability to use slices and minimize the size of the network on which an invariant is verified is important. Figure 5 shows time taken to verify the entire network as we increase the number of policy equivalence classes.

5.3 Other Network Scenarios

We next apply VMN to several other scenarios that illustrate the value of slicing (and symmetry) in reducing verification time.

5.3.1 Enterprise Network with Firewall

First, we consider a typical enterprise or university network protected by a stateful firewall, shown in Figure 6. The network interconnects three types of end hosts:

1. Hosts in *public* subnets should be allowed to both initiate and accept connections with the outside world.
2. Hosts in *private* subnets should be flow-isolated (*i.e.,* allowed to initiate connections to the outside world, but never accept incoming connections).
3. Hosts in *quarantined* subnets should be node-isolated (*i.e.,* not allowed to communicate with the outside world).

We vary the number of subnets keeping the proportion of subnet types fixed; a third of the subnets are public, a third are private and a third are quarantined.

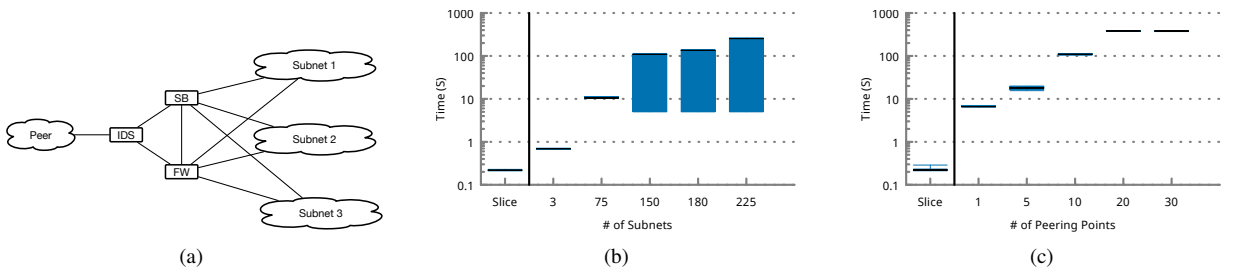
We configure the firewall so as to enforce the target invariants correctly: with two rules denying access (in either direction) for each quarantined subnet, plus one rule denying inbound connections for each private subnet. The results we present below are for the case where all the target invariants hold. Since this network only contains a firewall, using slices we can verify invariants on a slice whose size is independent of network size and policy complexity. We can also leverage the symmetry in both network and policy to reduce the number of invariants that need to be verified for the network. In contrast, when slices and symmetry are not used, the model for verifying each invariant grows as the size of the network, and we have to verify many more invariants. In Figure 7 we show time taken to verify the invariant using slices (Slice) and how verification time varies with network size when slices are not used.

5.3.2 Multi-Tenant Datacenter

Next, we consider how VMN can be used by a cloud provider (*e.g.,* Amazon) to verify isolation in a multi-tenant datacenter. We assume that the datacenter implements the Amazon EC2 Security Groups model [1]. For our test we considered a datacenter with 600 physical servers (which each run a virtual switch) and 210 physical switches (which implement equal cost multi-path routing). Tenants launch virtual machines (VMs), which are run on physical servers and connect to the network through virtual switches. Each virtual switch acts as a stateful firewall, and defaults to denying all traffic (*i.e.,* packets not specifically allowed by an ACL are dropped). To scale policy enforcement, VMs are organized in security groups with associated accept/-deny rules. For our evaluation, we considered a case where each tenant organizes their VMs into two security groups:

1. VMs that belong to the *public* security group are allowed to accept connections from any VMs.
2. VMs that belong to the *private* security group are flow-isolated (*i.e.,* they can initiate connections to other tenants' VMs, but can only accept connections from this tenant's public and private VMs).

We also assume that firewall configuration is specified in terms of security groups (*i.e.,* on receiving a packet the firewall computes the security group to which the sender and receiver belong and applies ACLs appropriately). For this evaluation, we configured the network to correctly enforce tenant policies. We added two ACL rules for each tenant's public security group allowing incoming and outgoing packets to anyone, while we added three rules for private security groups; two allowing incoming and outgoing traffic from the tenant's VM, and one allowing outgoing traffic to other tenants. For our evaluation we consider a case where each tenant has 10 VMs, 5 public and 5 private, which are spread across physical servers. These rules result in flow-parallel middleboxes, so we can use fixed size slices to verify each invariant. The number of invariants that need to be verified grow as a function of the number of tenants. In Figure 8 we show time taken to verify one instance of the invariant when slices are used (Slice) and how verification time varies with network size when



(a) (b) (c)

Figure 9: (a) shows the pipeline at each peering point for an ISP; (b) distribution of time to verify each invariant given this pipeline when the ISP peers with other networks at 5 locations; (c) average time to verify each invariant when the ISP has 75 subnets. In both cases, to the left of the black line we show time to verify on a slice (which is independent of network size) and vary sizes to the right.

slices are not used. The invariants checked are: (a) private hosts in one group cannot reach private hosts in another group (Priv-Priv), (b) public hosts in one group cannot reach private hosts in another group (Priv-Pub), and (c) private hosts in one group *can* reach public hosts in another.

5.3.3 ISP with Intrusion Detection

Finally, we consider an Internet Service Provider (ISP) that implements an intrusion detection system (IDS). We model our network on the SWITCHlan backbone [53], and assume that there is an IDS box and a stateful firewall at each peering point (Figure 9(a)). The ISP contains public, private and quarantined subnets (with policies as defined in §5.3.1) and the stateful firewalls enforce the corresponding invariants. Additionally, each IDS performs lightweight monitoring (*e.g.,* based on packet or byte counters) and checks whether a particular destination prefix (*e.g.,* a customer of the ISP) might be under attack; if so, all traffic to this prefix is rerouted to a scrubbing box that performs more heavyweight analysis, discards any part of the traffic that it identifies as "attack traffic," and forwards the rest to the intended destination. This combination of multiple lightweight IDS boxes and one (or a few) centralized scrubbing boxes is standard practice in ISPs that offer attack protection to their customers.[5]

To enforce the target invariants (for public, private, and quarantined subnets) correctly, all inbound traffic must go through at least one stateful firewall. We consider a misconfiguration where traffic rerouted by a given IDS box to the scrubbing box bypasses all stateful firewalls. As a result, any part of this rerouted traffic that is *not* discarded by the scrubbing box can reach private or quarantined subnets, violating the (simple or flow-) isolation of the corresponding end hosts.

When verifying invariants in a slice we again take advantage of the fact that firewalls and IDSes are flow-parallel. For each subnet, we can verify invariants in a slice containing a peering point, a end host from the subnet, the appropriate firewall, IDS and a scrubber. Furthermore, since all subnets belong to one of three policy equivalence classes, and the network is symmetric, we only need run verification on three slices.

We begin by evaluating a case where the ISP, similar to the SWITCHlan backbone has 5 peering points with other networks. We measure verification time as we vary the number of subnets (Figure 9(b)), and report time taken, on average, to verify each invariant. When slices are used, the median time for verifying an invariant is 0.21 seconds, by contrast when verification is performed on the entire network, a network with 250 subnets takes approximately 6 minutes to verify. Furthermore, when verifying all invariants, only 3 slices need to be verified when we account for symmetry, otherwise the number of invariants verified grows with network size.

In Figure 9(c) we hold the number of subnets constant (at 75) and show verification time as we vary the number of peering points. In this case the complexity of verifying the entire network grows more quickly (because the IDS model is more complex leading to a larger increase in problem size). In this case, verifying correctness for a network with 50 peering points, when verification is performed on the whole entire network, takes approximately 10 minutes. Hence, being able to verify slices and use symmetry is crucial when verifying such networks.

6 Related Work

Next, we discuss related work in network verification and formal methods.

Testing Networks with Middleboxes The work most closely related to us is Buzz [14], which uses symbolic execution to generate sequences of packets that can be used to test whether a network enforces an invariant. Testing, as provided by Buzz, is complimentary to verification. Our verification process does not require sending traffic through the network, and hence provides a non-disruptive mechanism for ensuring that changes to a network (*i.e.,* changing middlebox or routing configuration, adding new types of middleboxes, etc.) do not result in invariant violation. Verification is also useful when initially designing a network, since designs can be evaluated to ensure they uphold desirable invariants. However, as we have noted,

[5] This setup is preferred to installing a separate scrubbing box at each peering point because of the high cost of these boxes, which can amount to several million dollars for a warranteed period of 3 years.

our verification results hold if and only if middlebox implementations are correct, *i.e.,* packets are correctly classified, etc. Combining a verification tool like VMN with a testing tool such as Buzz allows us to circumvent this problem, when possible (*i.e.,* when the network is not heavily utilized, or when adding new types of middleboxes), Buzz can be used to test if invariants hold. This is similar to the relationship between ATPG (testing) and HSA (verification), and more generally to the complimentary use of verification and testing in software development today.

Beyond the difference of purpose, there are some other crucial difference between Buzz and VMN: (a) in contrast to VMN, Buzz's middlebox models are specialized to a context and cannot be reused across networks, and (b) Buzz does not use techniques such as slicing, limiting its applicability to networks with only several 100 nodes. We believe our slicing techniques can be adopted to Buzz.

Similarly, SymNet [51] proposes the use of symbolic execution for verifying network reachability. They rely on models written in a symbolic execution friendly language SEFL where models are supposed to closely resemble middlebox code. However, to ensure feasibility for symbolic execution, SEFL does not allow unbounded loops, or pointer arithmetic and limits access to semantically meaningful packet fields. These models are therefore very different from the implementation for high performance middleboxes. Furthermore, due to these limitations SEFL cannot model middleboxes like IDSes and IPSes, and is limited to modeling *flow-parallel* middleboxes.

Verifying Forwarding Rules Recent efforts in network verification [2, 5, 17, 27, 28, 35, 48, 50] have focused on verifying the network dataplane by analyzing forwarding tables. Some of these tools including HSA [26], Libra [60] and VeriFlow [28] have also developed algorithms to perform near real-time verification of simple properties such as loop-freedom and the absence of blackholes. Recent work [40] has also shown how techniques similar to slicing can be used to scale these techniques. Our approach generalizes this work by accounting for state and thus extends verification to mutable datapaths.

Verifying Network Updates Another line of network verification research has focused on verification during configuration updates. This line of work can be used to verify the consistency of routing tables generated by SDN controllers [25, 55]. Recent efforts [34] have generalized these mechanisms and can determine what parts of configuration are affected by an update, and verify invariants on this subset of the configuration. This work does not consider dynamic, stareful datapath elements with more frequent state updates.

Verifying Network Applications Other work has looked at verifying the correctness of control and data plane applications. NICE [5] proposed using static analysis to verify the correctness of controller programs. Later extensions including [31] have looked at improving the accuracy of NICE using concolic testing [47] by trading away completeness. More recently, Vericon [4] has looked at sound verification of a restricted class of controllers.

Recent work [10] has also looked at using symbolic execution to prove properties for programmable datapaths (middleboxes). This work in particular looked at verifying bounded execution, crash freedom and that certain packets are filtered for stateless or simple stateful middleboxes written as pipelines and meeting certain criterion. The verification technique does not scale to middleboxes like content caches which maintain arbitrary state.

Finite State Model Checking Finite state model checking has been applied to check the correctness of many hardware and software based systems [5, 7, 27]. Here the behavior of a system is specified as a transition relation between finite state and a checker can verify that all reachable states from a starting configuration are safe (*i.e.,* do not cause any invariant violation). However these techniques scale exponentially with the number of states and for even moderately large problems one must choose between being able to verify in reasonable time and completeness. Our use of SMT solvers allows us to reason about potentially infinite state and our use of simple logic allows verification to terminate in a decidable manner for practical networks.

Language Abstractions Several recent works in software-defined networking [16, 21, 29, 36, 58] have proposed the use of verification friendly languages for controllers. One could similarly extend this concept to provide a verification friendly data plane language however our approach is orthogonal to such a development: we aim at proving network wide properties rather than properties for individual middleboxes

7 Conclusion

In this paper we presented VMN, a system for verifying isolation invariants in networks with middleboxes. The key insights behind our work is that abstract middleboxes are well suited to verifying network configuration; and the use of slices which allow invariants to be verified on a subset of the network is essential to scaling.

8 Acknowledgment

We thank Shivaram Venkatraman, Colin Scott, Kay Ousterhout, Amin Tootoonchian, Justine Sherry, Sylvia Ratnasamy, Nate Foster, the anonymous reviewers and our shepherd Boon Thau Loo for the many helpful comments that have greatly improved both the techniques described within this paper and their presentation. This work was supported in part by a grant from Intel Corporation, NSF grant 1420064, and an ERC grant (agreement number 321174-VSSC).

References

[1] AMAZON. Amazon EC2 Security Groups. http://goo.gl/GfYQmJ, Oct. 2014.

[2] ANDERSON, C. J., FOSTER, N., GUHA, A., JEANNIN, J.-B., KOZEN, D., SCHLESINGER, C., AND WALKER, D. NetKAT: Semantic foundations for networks. In *POPL* (2014).

[3] Azure Storage. Retrieved 01/23/2016 https://azure.microsoft.com/en-us/services/storage/.

[4] BALL, T., BJØRNER, N., GEMBER, A., ITZHAKY, S., KARBYSHEV, A., SAGIV, M., SCHAPIRA, M., AND VALADARSKY, A. VeriCon: Towards Verifying Controller Programs in Software-Defined Networks. In *PLDI* (2014).

[5] CANINI, M., VENZANO, D., PERES, P., KOSTIC, D., AND REXFORD, J. A NICE Way to Test OpenFlow Applications. In *NSDI* (2012).

[6] CARPENTER, B., AND BRIM, S. RFC 3234: Middleboxes: Taxonomy and Issues, Feb. 2002.

[7] CLARKE, E. M., GRUMBERG, O., AND PELED, D. *Model checking*. MIT Press, 2001.

[8] DATACENTER WORLD. Largest Data Centers: i/o Data Centers, Microsoft. https://goo.gl/dB9VdO retrieved 09/18/2016, 2014.

[9] DE MOURA, L., AND BJØRNER, N. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008.

[10] DOBRESCU, M., AND ARGYRAKI, K. Software Dataplane Verification. In *NSDI* (2014).

[11] EISENBUD, D. E., YI, C., CONTAVALLI, C., SMITH, C., KONONOV, R., MANN-HIELSCHER, E., CILINGIROGLU, A., CHEYNEY, B., SHANG, W., AND HOSEIN, J. D. Maglev: A fast and reliable software network load balancer. In *NSDI* (2016).

[12] EMERGING THREATS. Emerging Threats open rule set. https://rules.emergingthreats.net/ retrieved 09/18/2016, 2016.

[13] ETSI. Network Functions Virtualisation. Retrieved 07/30/2014 http://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[14] FAYAZ, S. K., YU, T., TOBIOKA, Y., CHAKI, S., AND SEKAR, V. BUZZ: Testing Context-Dependent Policies in Stateful Networks. In *NSDI* (2016).

[15] FORWARD NETWORKS. Forward Networks. https://www.forwardnetworks.com/, 2016.

[16] FOSTER, N., GUHA, A., REITBLATT, M., STORY, A., FREEDMAN, M. J., KATTA, N. P., MONSANTO, C., REICH, J., REXFORD, J., SCHLESINGER, C., WALKER, D., AND HARRISON, R. Languages for software-defined networks. *IEEE Communications Magazine 51*, 2 (2013), 128–134.

[17] FOSTER, N., KOZEN, D., MILANO, M., SILVA, A., AND THOMPSON, L. A Coalgebraic Decision Procedure for NetKAT. In *POPL* (2015).

[18] GADDE, S., CHASE, J., AND RABINOVICH, M. A Taste of Crispy Squid. In *Workshop on Internet Server Performance* (1998).

[19] Amazon Glacier. Retrieved 01/23/2016 https://aws.amazon.com/glacier/.

[20] GRAZIANO, P. Speed Up Your Web Site with Varnish. *Linux Journal 2013* (Mar. 2013).

[21] GUHA, A., REITBLATT, M., AND FOSTER, N. Machine-verified network controllers. In *PLDI* (2013), pp. 483–494.

[22] INTEL. Data Plane Develpment Kit. http://dpdk.org/, 2016.

[23] ITZHAKY, S., BANERJEE, A., IMMERMAN, N., LAHAV, O., NANEVSKI, A., AND SAGIV, M. Modular reasoning about heap paths via effectively propositional formulas. In *POPL* (2014).

[24] JUNKER, H. OWASP Enterprise Security API. *Datenschutz und Datensicherheit 36* (2012), 801–804.

[25] KATTA, N. P., REXFORD, J., AND WALKER, D. Incremental consistent updates. In *NSDI* (2013).

[26] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *NSDI* (2013).

[27] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: Static checking for networks. In *NSDI* (2012).

[28] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. Veriflow: Verifying network-wide invariants in real time. In *NSDI* (2013).

[29] KOPONEN, T., AMIDON, K., BALLAND, P., CASADO, M., CHANDA, A., FULTON, B., GANICHEV, I., GROSS, J., GUDE, N., INGRAM, P., JACKSON, E., LAMBETH, A., LENGLET, R., LI, S.-H., PADMANABHAN, A., PETTIT, J., PFAFF, B., RAMANATHAN, R., SHENKER, S., SHIEH, A., STRIBLING, J., THAKKAR, P., WENDLANDT, D., YIP, A., AND ZHANG, R. Network virtualization in multi-tenant datacenters. In *NSDI* (2014).

[30] KOROVIN, K. Non-cyclic sorts for first-order satisfiability. In *Frontiers of Combining Systems*, P. Fontaine, C. Ringeissen, and R. Schmidt, Eds., vol. 8152 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 214–228.

[31] KUZNIAR, M., PERESINI, P., CANINI, M., VENZANO, D., AND KOSTIC, D. A SOFT Way for OpenFlow Switch Interoperability Testing. In *CoNEXT* (2012).

[32] LEBLOND, E. Secure use of iptables and connection tracking helpers. `https://goo.gl/wENPDy` retrieved 09/18/2016, 2012.

[33] LICHTENSTEIN, O., PNUELI, A., AND ZUCK, L. D. The glory of the past. In *Logics of Programs, Conference, Brooklyn College, June 17-19, 1985, Proceedings* (1985), pp. 196–218.

[34] LOPES, N. P., BJØRNER, N., GODEFROID, P., JAYARAMAN, K., AND VARGHESE, G. DNA Pairing: Using Differential Network Analysis to find Reachability Bugs. Tech. rep., Microsoft Research, 2014. research.microsoft.com/pubs/215431/paper.pdf.

[35] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P., AND KING, S. T. Debugging the data plane with Anteater. In *SIGCOMM* (2011).

[36] NELSON, T., FERGUSON, A. D., SCHEER, M. J. G., AND KRISHNAMURTHI, S. A balance of power: Expressive, analyzable controller programming. In *NSDI* (2014).

[37] PASQUIER, T., AND POWLES, J. Expressing and enforcing location requirements in the cloud using information flow control. In *International Workshop on Legal and Technical Issues in Cloud Computing (CLaw). IEEE* (2015).

[38] PEREIRA, H., RIBEIRO, A., AND CARVALHO, P. L7 classification and policing in the pfsense platform. *Atas da CRC* (2009).

[39] PISKAC, R., DE MOURA, L. M., AND BJØRNER, N. Deciding Effectively Propositional Logic using DPLL and substitution sets. *J. Autom. Reasoning 44*, 4 (2010).

[40] PLOTKIN, G. D., BJØRNER, N., LOPES, N. P., RYBALCHENKO, A., AND VARGHESE, G. Scaling network verification using symmetry and surgery. In *POPL* (2016).

[41] POTHARAJU, R., AND JAIN, N. Demystifying the dark side of the middle: a field study of middlebox failures in datacenters. In *IMC* (2013).

[42] PURDY, G. N. *Linux iptables - pocket reference: firewalls, NAT and accounting*. 2004.

[43] QUALYS INC. IronBee. `http://ironbee.com/` retrieved 09/18/2016, 2016.

[44] RISTIC, I. *ModSecurity Handbook*. 2010.

[45] ROESCH, M. Snort - Lightweight Intrusion Detection for Networks. In *LISA* (1999).

[46] Amazon S3. Retrieved 01/23/2016 `https://aws.amazon.com/s3/`.

[47] SEN, K., AND AGHA, G. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV* (2006).

[48] SETHI, D., NARAYANA, S., AND MALIK, S. Abstractions for Model Checking SDN Controllers. In *FMCAD* (2013).

[49] SHERRY, J., HASAN, S., SCOTT, C., KRISHNAMURTHY, A., RATNASAMY, S., AND SEKAR, V. Making middleboxes someone else's problem: Network processing as a cloud service. In *SIGCOMM* (2012).

[50] SKOWYRA, R., LAPETS, A., BESTAVROS, A., AND KFOURY, A. A Verification Platform for SDN-Enabled Applications. In *HiCoNS* (2013).

[51] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. SymNet: scalable symbolic execution for modern networks. *CoRR abs/1604.02847* (2016).

[52] STONESOFT. StoneGate Administrator's Guide v5.3. `https://goo.gl/WNcaQj` retrieved 09/18/2016, 2011.

[53] SWITCH. SWITCHlan Backbone. `http://goo.gl/iWm9VE`.

[54] TARREAU, W. HAProxy-the reliable, high-performance TCP/HTTP load balancer. `http://haproxy.lwt.eu`, 2012.

[55] VANBEVER, L., REICH, J., BENSON, T., FOSTER, N., AND REXFORD, J. HotSwap: Correct and Efficient Controller Upgrades for Software-Dfined Networks. In *HOTSDN* (2013).

[56] VELNER, Y., ALPERNAS, K., PANDA, A., RABINOVICH, A. M., SAGIV, S., SHENKER, S., AND SHOHAM, S. Some complexity results for stateful network verification. In *TACAS* (2016).

[57] VERIFLOW INC. Veriflow. `http://www.veriflow.net/`, 2016.

[58] VOELLMY, A., WANG, J., YANG, Y. R., FORD, B., AND HUDAK, P. Maple: simplifying sdn programming using algorithmic policies. In *SIGCOMM* (2013).

[59] ZELDOVICH, N., BOYD-WICKIZER, S., AND MAZIÈRES, D. Securing distributed systems with information flow control. In *NSDI* (2008).

[60] ZENG, H., ZHANG, S., YE, F., JEYAKUMAR, V., JU, M., LIU, J., MCKEOWN, N., AND VAHDAT, A. Libra: Divide and conquer to verify forwarding tables in huge networks. In *NSDI* (2014).

A Real-world Models

Here we provide models for real world middleboxes, in addition to the ones listed in §2.3.

NATs We also examined the NAT functions of `iptables` and pfSense. Each of them provides both a "source NAT" (SNAT) function, which allows end-hosts with private addresses to initiate Internet connections, and a "destination NAT" (DNAT) function, which allows end-hosts with private addresses to accept Internet connections.

Listing 4 shows a VMN model for both SNATs. The configuration input is the box's public address (Line 1). The `remapped_port` oracle returns an available port to be used for a new connection, abstracting away the details of how the port is chosen (Line 2); during verification, we assume that the oracle may return any port. The `active` and `reverse` maps associate private addresses to ports and vice versa (Lines 3–4). The forwarding model: on receiving a packet that belongs to an established connection, the necessary state is retrieved from either the `reverse` map—when the packet comes from the public network (Lines 6–10)—or the `active` map—when the packet comes from the private network (Lines 11–14); on receiving a packet from the private network that is establishing a new connection, the oracle is consulted to obtain a new port (Line 19) and the relevant state is recorded in the maps (Lines 20–21) before the packet is forwarded appropriately (Line 22). To model a SNAT that uses a pool of public addresses (as opposed to a single address), we instantiate one SNAT object (as defined in Listing 4) per address and define an oracle that returns which SNAT object to use for each connection.

Listing 4: Model for a Source NAT

```
1  class SNAT (nat_address: Address){
2    abstract
       ↪ remapped_port (p: Packet): int
3    val active : Map[Flow, int]
4    val reverse
       ↪ : Map[port, (Address, int)]
5    def model (p: Packet) = {
6      dst(p) == nat_address =>
7        (dst, port)
           ↪ = reverse[p.dst_port];
8        p.dst = dst;
9        p.dst_port = port;
10       forward(Seq(p))
11     active.contains(flow(p)) =>
12       p.src = nat_address;
13       p.src_port = active(flow(p));
14       forward(Seq(p))
15     _ =>
16       address = p.src;
17       port = p.src_port
18       p.src = nat_address;
19       p.src_port = remapped_port(p);
20       active(flow(p)) = p.src_port;
21       reverse(p.src_port)
           ↪ = (address, port);
22       forward(Seq(p))
23   }
24 }
```

Listing 5 shows a VMN model for both DNATs. The configuration input is a map associating public address/port pairs to private ones (Line 1). There are no oracles. The `reverse` map associates private address/port pairs to public ones (Line 2). The forwarding model: on receiving, from the public network, a packet whose destination address/port were specified in the configuration input, the packet header is updated accordingly and the original destination address/port pair recorded in the `reverse` map (Lines 3–9); conversely, on receiving, from the private network, a packet that belongs to an established connection, the necessary state is retrieved from the `reverse` map and the packet updated accordingly (Lines 10–13); any other received packets pass through unchanged.

Listing 5: Model for a Destination NAT

```
1  class DNAT(translations:
     ↪ Map[(Address,
     ↪ int), (Address, int)]) {
2    val reverse:
       ↪ Map[Flow, (Address, int)]
3    def model (p: Packet) = {
4      translations.contains((p.dst,
         ↪ p.dst_port)) =>
5        dst = p.dst;
6        dst_port = p.dst_port;
7        p.dst = translations[(p.dst,
             ↪ p.dst_port)]._1;
8        p.dst_port
             ↪ = translations[(p.dst,
```

```
            ↪ p.dst_port)]._2;
        reverse[flow(p)]
            ↪ = (dst, dst_port);
        forward(Seq(p))
      reverse.contains(flow(p)) =>
        p.src = reverse[flow(p)]._1;
        p.src_port
            ↪ = reverse[flow(p)]._2;
        forward(Seq(p))
      _ => forward(Seq(p))
    }
}
```

Gateways A network gateway often performs firewall, NAT, and/or IDS/IPS functions organized in a chain. In fact, both `iptables` and pfSense are modular gateways that consist of configurable chains of such functions (*e.g.,* in `iptables` one can specify a `CHAIN` parameter for NAT rules). To model such a modular gateway, we have written a script that reads the gateway's configuration and creates a pipeline of models, one for each specified function.

Listing 6: Model for a Load Balancer

```
class LoadBalancer(backends:
    ↪ List[Address]) {
  val assigned: Map[Flow, Address]
  abstract
      ↪ pick_backend(p: Packet): int
  def model (p: Packet) = {
    assigned.contains(flow(p)) =>
      p.dst = assigned[flow(p)]
      forward(Seq(p))
    _ =>
      assigned[flow(p)] =
          ↪ backends[pick_backend(p)]
      p.dst = assigned[flow(p)]
      forward(Seq(p))
  }
}
```

Load-balancers A load-balancer performs a conceptually simple function: choose the backend server that will handle each new connection and send to it all packets that belong to the connection. We considered two kinds: systems like HAProxy [54], which control the backend server that will handle a packet by rewriting the packet's destination address, and systems like Maglev [11], which control the backend server through the packet's output network interface. Both kinds accept as configuration the set of available backend servers (either their addresses or the network interface of the load balancer that leads to each server). Our load-balancer (Listing 6) uses an oracle to determine which server handles a connection, during verification the decission process can therefore choose from any of the available servers.

Listing 7: Model for a simple cache

```
class Cache(address: Address, acls:
    ↪ Set[(Address, Address)]) {
 abstract request(p: Packet): int
 abstract response(p: Packet): int
 abstract new_port(p: Packet): int
 abstract cacheable(int): bool
 val outstanding_requests:
     ↪ Map[Flow, int]
 val outstanding_conns:
     ↪ Map[Flow, Flow]
 val cache_origin: Map[int, Host]
 val origin_addr: Map[int, Address]
 val cached: Map[int, int]
 def model (p: Packet) = {
  val p_flow = flow(p);
  outstanding_request.contains(p_flow)
      ↪ &&
  cacheable(
    outstanding_request[p_flow]) =>
    cached[outstanding_request[p_flow]]
            = response(p);
    ...
    p.src =
      outstanding_conns[p_flow].src;
    p.dst =
      outstanding_conns[p_flow].dst;
    ...
    forward(Seq(p))
  outstanding_request.contains(p_flow)
      ↪ &&
  !cacheable(
     outstanding_request[p_flow]] =>
    p.src =
      outstanding_conns[p_flow].src;
    ...
    forward(Seq(p))
  cached.contains(request(p)) &&
  !acls.contains(
    p.src, origin_addr[request(p)]) =>
      p.src = p_flow.dst;
      ...
      p.origin =
          ↪ cache_origin[request(p)];
      forward(Seq(p))
  !acls.contains(p.src, p.dst) =>
      p.src = address;
      p.src_port = new_port(p);
      outstanding_conns[flow(p)]
          ↪ = p_flow;
      outstanding_requests[flow(p)]
          ↪ = request(p)
      forward(Seq(p))
  _ =>
      forward(Empty)
  }
}
```

Caches We examined two caches: Squid [18] and Apache Web Proxy. These systems have a rich set of config-

uration parameters that control, *e.g.,* the duration for which a cached object is valid, the amount of memory that can be used, how DNS requests are handled, etc. However, most of them are orthogonal to our invariants. We therefore consider only a small subset of the configuration parameters, in particular, those that determine whether a request should be cached or not, and who has access to cached content.

Listing 7 shows a model that captures both caches. Configuration input is a list of rules specifying which hosts have access to content originating at particular servers (Line 1). We define four oracles, among them `cacheable`, which determines whether policy and cache-size limits allow a response to be cached (Line 5). The forwarding model captures the following behavior: on receiving a request for content that is permitted by configuration, we check whether this content has been cached (Line 32–38); if so, we respond to the request, otherwise we forward the request to the corresponding server (Line 39–44); on receiving a response from a server, we check if the corresponding content is cacheable, if so, we cache it (Line 15–24); and regardless of its cacheability forward the response to the client who originally requested this content.

We treat each request and response as a single packet, whereas, in reality, they may span multiple packets. This greatly simplifies the model and—since we do not check performance-related invariants—does not affect verification results.

B Logical Models

We model network behavior in discrete timesteps. During each timestep a previously sent packet can be delivered to a node (middlebox or host), a host can generate a new packet that enters the network, or a middlebox can process a previously received packet. We do not attempt to model the *likely* order of these various events, but instead consider all such orders in search of invariant violations. In this case Z3 acts as a *scheduling oracle* that assigns an event to each timestep, subject to the standard causality constraints, *i.e.,* we ensure that packets cannot be received before being sent, and packets sent on the same link are not reordered.

VMN models middleboxes and networks using quantified logical formula, which are axioms describing how received packets are treated. Oracles in VMN are modeled as uninterpreted function, *i.e.,* Z3 can assign any (convinient) value to a given input to an oracle. We also provide Z3 with the negation of the invariant to be verified, which is specified in terms of sets of packets (or data) that are sent or received. Finding a satisfiable assignment to these formulae is equivalent to Z3 finding a set of oracle behaviors that result in the invariant being violated, and proving the formulae unsatisfiable is equivalent to showing that no oracular behavior can result in the invariants being violated.

Symbol	Meaning
Events	
$rcv(d,s,p)$	Destination d receives packet p from source s.
$snd(s,d,p)$	Source s sends packet p to destination d.
Logical Operators	
$\Box P$	Condition P holds at all times.
$\blacklozenge P$	Event P occurred in the past.
$\neg P$	Condition P does not hold (or event P does not occur).
$P_1 \wedge P_2$	Both conditions P_1 and P_2 hold.
$P_1 \vee P_2$	Either condition P_1 or P_2 holds.

Table 1: Logical symbols and their interpretation.

B.1 Notation

We begin by presenting the notation used in this section. We express our models and invariants using a simplified form of linear temporal logic (LTL) [33] of events, with past operators. We restrict ourselves to safety properties, and hence only need to model events occurring in the past or events that hold globally for all of time. We use LTL for ease of presentation; VMN converts LTL formulae to first-order formulas (required by Z3) by explicitly quantifying over time. Table 1 lists the symbols used to express formula in this section.

Our formulas are expressed in terms of three events: $snd(s,d,p)$, the event where a *node* (end host, switch or middlebox) s sends *packet* p to *node* d; and $rcv(d,s,p)$, the event where a node d receives a packet p from node s, and $fail(n)$, the event where a node n has failed. Each event happens at a timestep and logical formulas can refer either to events that occurred in the past (represented using $\blacklozenge$) or properties that hold at all times (represented using $\Box$). For example,

$$\forall d,s,p\colon \Box(rcv(d,s,p) \implies \blacklozenge snd(s,d,p))$$

says that at all times, any packet p received by node d from node s must have been sent by s in the past.

Similarly,

$$\forall p\colon \Box\neg rcv(d,s,p)$$

indicates that d will never receive any packet from s.

Header fields and oracles are represented using functions, *e.g., src*(p) and $dst(p)$ represent the source and destination address for packet p, and $mallicious(p)$ acts as the `mallicious` oracle from Listing 1.

B.2 Reachability Invariants

Reachability invariants can be be generally specifies as:

$$\forall n,p\colon \Box\neg(rcv(d,n,p) \wedge predicate(p)),$$

which says that node d should never receive a packet p that matches $predicate(p)$. The *predicate* can be expressed in terms of packet-header fields, abstract packet classes and past events, this allows us to express a wide variety

Listing 8: Model for a learning firewall

```
class LearningFirewall (acl:
    ↪ Set[(Address, Address)]) {
  val established : Set[Flow]
  def model (p: Packet) = {
    established.contains(flow(p)) =>
      forward (Seq(p))
    acl.contains((p.src, p.dest)) =>
      established += flow(p)
      forward(Seq(p))
    _ =>
      forward(Seq.empty)
  }
}
```

of network properties as reachability invariants, *e.g.,*:

• Simple isolation: node d should never receive a packet with source address s. We express this invariant using the *src* function, which extracts the source IP address from the packet header:

$$\forall n,p : \Box\neg(rcv(d,n,p) \wedge src(p) = s).$$

• Flow isolation: node d can only receive packets from s if they belong to a previously established flow. We express this invariant using the $flow$ function, which computes a flow identifier based on the packet header:

$$\forall n_0,p_0,n_1,p_1 : \Box\neg(rcv(d,n_0,p_0) \wedge src(p_0) = s \wedge \neg(\blacklozenge snd(d,n_1,p_1) \wedge flow(p_1) = flow(p_0))).$$

• Data isolation: node d cannot access any data originating at server s, this requires that d should not access data either by directly contacting s or indirectly through network elements such as content cache. We express this invariant using an *origin* function, that computes the origin of a packet's data based on the packet header (*e.g.,* using the `x-http-forwarded-for` field in HTTP):

$$\forall n,p : \Box\neg(rcv(d,n,p) \wedge origin(p) = s).$$

B.3 Modeling Middleboxes

Middleboxes in VMN are specified using a high-level loop-free, event driven language. Restricting the language so it is loop free allows us to ensure that middlebox models are expressible in first-order logic (and can serve as input to Z3). We use the event-driven structure to translate this code to logical formulae (axioms) encoding middlebox behavior.

VMN translates these high-level specifications into a set of parametrized axioms (the parameters allow more than one instance of the same middlebox to be used in a network). For instance, Listing 8 results in the following axioms:

$$\textbf{established}(flow(p)) \implies (\blacklozenge((\neg fail(\mathbf{f})) \wedge (\blacklozenge rcv(\mathbf{f},p)))) \wedge \textbf{acl}(src(p),dst(p))$$

$$send(\mathbf{f},p) \implies (\blacklozenge rcv(\mathbf{f},p)) \wedge (\textbf{acl}(src(p),dst(p)) \vee \textbf{established}(flow(p)))$$

The bold-faced terms in this axiom are parameters: for each stateful firewall that appears in a network, VMN adds a new axiom by replacing the terms **f**, **acl** and **established** with a new instance specific term. The first axiom says that the **established** set contains a flow if a packet permitted by the firewall policy (**acl**) has been received by **f** since it last failed. The second one states that packets sent by **f** must have been previously received by it, and are either pr emitted by the **acl**'s for that firewall, or belong to a previously established connection.

We translate models to formulas by finding the set of packets appearing in the `forward` function appearing at the end of each match statement, and translating the statement so that the middlebox sending that set of packet implies that (a) previously a packet matching an appropriate criterion was received; and (b) middlebox state was appropriately updated. We combine all branches where the same set of packets are updated using logical conjunction, *i.e.,* implying that one of the branches was taken.

B.4 Modeling Networks

VMN uses *transfer functions* to specify a network's forwarding behavior. The transfer function for a network is a function from a located packet to a set of located packets indicating where the packets are next sent. For example, the transfer function for a network with 3 hosts A (with IP address a), B (with IP address b) and C (with IP address c) is given by:

$$f(p,port) \equiv \begin{cases} (p,A) & \text{if } dst(p) = a \\ (p,B) & \text{if } dst(p) = b \\ (p,C) & \text{if } dst(p) = c \end{cases}$$

VMN translates such a transfer function to axioms by introducing a single pseudo-node (Ω) representing the network, and deriving a set of axioms for this pseudo-node from the transfer function and failure scenario. For example, the previous transfer function is translated to the following axioms ($fail(X)$ here represents the specified failure model).

$$\forall n,p : \Box fail(X) \wedge ... snd(A,n,p) \implies n = \Omega$$

$$\forall n,p : \Box fail(X) \wedge ... snd(\Omega,n,p) \wedge dst(p) = a \implies n = A \wedge \blacklozenge \exists n' : rcv(n',\Omega,p)$$

In addition to the axioms for middlebox behavior and network behavior, VMN also adds axioms restricting the oracles' behavior, *e.g.,* we add axioms to ensure that any packet delivery event scheduled by the scheduling oracle has a corresponding packet send event, and we ensure that new packets generated by hosts are well formed.

C Formal Definition of Slices

Given a network $N = (V, E, P)$, with network transfer function N_T, we define a subnetwork Ω to be the network formed by taking a subset $V|_\Omega \subseteq \Omega$ of nodes, all the links connecting nodes in $V|_\Omega$ and a subset of packets $P|_\Omega \subseteq P$ from the original network. We define a subnetwork Ω to be a slice if and only if Ω is closed under *forwarding* and *state*, and $P|_\Omega$ is maximal. We define $P|_\Omega$ to be maximal if and only if $P|_\Omega$ includes all packets from P whose source and destination are in $V|_\Omega$.

We define a subnetwork to be *closed under forwarding* if and only if (a) all packets $p \in P|_\Omega$ are located inside Ω, *i.e.,* $p.loc \in V|_\Omega \forall p \in P|_\Omega$; and (b) the network transfer function forwards packets within Ω, *i.e.,* $N_T(p) \subseteq P|_\Omega \forall p \in P|_\Omega$.

The definition for being *closed under state* is a bit more involved, informally it states that all states reachable by a middlebox in the network is also reachable in the slice. More formally, associate with the network a set of states S where each state $s \in S$ contains a multiset of pending packets $s.\Pi$ and the state of each middlebox ($s.m_0$ for middlebox m_0). Given this associated set of states we can treat the network as a state machine, where each transition is a result of one of two actions:

- An end host $e \in V$ generates a packet $p \in P$, in this case the system transitions to the state where all packets in $N_T(p)$ (where N_T is the network transfer function defined above) are added to the multiset of pending packets.
- A packet p contained in the pending state is delivered to $p.loc$. In cases where $p.loc$ is an end host, this merely results in a state where one p is removed from the multiset of pending packets. If however, $p.loc$ is a middlebox we transition to the state gotten by (a) removing p from pending packets, (b) updating the state for middlebox $p.loc$ and (c) for all packets p' forwarded by the middlebox, adding $N_T(p')$ to the set of pending packets.

In this model, invariants are predicates on states, an invariant is violated if and only if the system transitions to a state where the invariant is true.

Observe that this definition of state machines can be naturally restricted to apply to a subnetwork Ω that is closed under forwarding, by associating a set of states S_Ω containing the state only for those middleboxes in Ω. Finally, given some subnetwork Ω we define a restriction function σ_Ω that relates the state space for the whole network S to S_Ω the state space for the subnetwork. For any state $s \in S$, σ simply drops all packets not in $P|_\Omega$ and drops the state for all middleboxes $m \notin V|_\Omega$.

Finally, we define some state $s \in S$ as reachable in N if and only if there exists a sequence of actions starting from the initial state (where there are no packets pending and all middleboxes are set to their initial state) that results in network N getting to state s. A similar concept of reachability of course also applies to the state machine for Ω. Finally, we define a subnetwork Ω to be *closed under state* if and only if S_Ω, the set of states reachable in Ω is the same as the projection of the set of states reachable in the network, more formally $S_\Omega = \{\sigma_\Omega(s), s \in S, s \text{ reachable in } N\}$.

When a subnetwork Ω is closed under *forwarding* and *state*, one can establish a bisimulation between the slice and the network N; informally this implies that one can find a relation such that when we restrict ourselves to packets in $p \in P|_\Omega$ then all transitions in N have a corresponding transition in Ω, corresponding here implies that the states in N are the same as the states in Ω after projection. Since by definition for any slice $P|_\Omega$ the set of packets is maximal, this means that every state reachable in N has an equivalent projection in Ω.

Finally, we define an invariant I to be evaluable in a subnetwork Ω if and only if for all states $s_1, s_2 \in S$ $\sigma_\Omega(s_1) = \sigma_\Omega(s_2) \implies I(s_1) = I(s_2)$, *i.e.,* if the invariant does not depend on any state not captured by Ω. As a result of the bisimulation between network N and slice Ω, it is simple to see that an invariant I evaluable in Ω holds in network N if and only if it also holds in Ω. Thus once a slice is found, we can verify any invariants evaluable on it and trivially transfer the verification results to the whole network.

Note, that we can always find a slice of the network on which an invariant can be verified, this is trivially true since the network itself is its own slice. The challenge therefore lies in finding slices that are significantly smaller than the entire network, and of sizes that do not grow as more devices are added to the network. The nodes that are included in a slice used to verify an invariant trivially depend on the invariant being verified, since we require that the invariant be evaluable on the slice. However, since slices must be closed under state, their size is also dependent on the types of middleboxes present in the network. Verification for network where all middleboxes are such that their state can be partitioned (based on any criterion, *e.g.,* flows, policy groups, etc.) are particularly amenable to this approach for scaling. We present two concrete classes of middleboxes that contain all of the examples we have listed previously in §2.3 that allow verification to be performed on slices whose size is independent of the network's size.

D Decidability

As noted in §4.2, first-order logic is undecidable. Further, verifying a network with mutable datapaths is undecidable in general, such networks are Turing complete. However, we believe that we can express networks obeying the following restrictions in a decidable fragment of first-order logic:

1. All middleboxes used in the network are passive, *i.e.,* they send packets only in response to packets received by them. In particular this means that every packet sent by

a middlebox is causally related to some packet previously sent by an end-host.

2. A middlebox sends a finite number of packets in response to any packets it receives.

3. All packet processing pipelines are loop free, *i.e.,* any packets that enter the network are delivered or dropped in a finite number of steps.

We now show that we can express middleboxes and networks meeting this criterion in a logic built by extending "effectively propositional logic" (*EPR*). EPR is a fundamental, decidable fragment of first-order logic [39], where all axioms are of the form $\exists^*\forall^*$, and the invariant is of the form $\forall^*\exists^*$. Neither axioms nor invariants in this logic can contain function symbols. EPR-F extends EPR to allow some unary functions. To guarantee decidability, EPR-F requires that there exist a finite set of compositions of unary functions U, such that any composition of unary functions can be reduced to a composition in U. For example, when a single unary function f is used, we require that there exist k such that $f^k(x) = f^{k-1}(x)$ for all k. Function symbols that go from one type to another are allowed, as long as their inverse is not used [30] (*e.g.,* we can use *src* in our formulas since it has no inverse). Prior work [23] has discussed mechanisms to reduce EPR-F formulas to EPR.

We can translate our models to EPR-F by:

1. Reformulate our assertions with "event variables" and functions that assign properties like time, source and destination to an event. We use predicate function to mark events as either being sends or receives.

2. Replace $\forall\exists$ formulas with equivalent formulas that contain Skolem functions instead of symbols.

For example the statement

$$\forall d,s,p : rcv(d,s,p) \Longrightarrow \blacklozenge\, snd(s,d,p)$$

is translated to the formula

$$\forall e : rcv(e) \Rightarrow snd(cause(e)) \wedge \ldots \wedge t(cause(e)) < t(e)$$

We also add the axiom, $\forall e : snd(e) \Rightarrow cause(e) = e$ which says that a *snd* event has no cause, ensuring idempotency.

To show that our models are expressible in EPR-F, we need to show that all unary functions introduced during this conversion meet the required closure properties. Intuitively, all function introduced by us track the causality of network events. Our decidability criterion imply that every network event has a finite causal chain. This combined with the axiom that $cause(e)$ is idempotent implies that the functions meet the closure properties. However, for Z3 to guarantee termination, explicit axioms guaranteeing closure must be provided. Generating these axioms from a network is left to future work. In our experience, Z3 terminates on networks meeting our criterion even in the absence of closure axioms.

Automated Bug Removal for Software-Defined Networks

Yang Wu*, Ang Chen*, Andreas Haeberlen*, Wenchao Zhou†, Boon Thau Loo*

*University of Pennsylvania, †Georgetown University

Abstract

When debugging an SDN application, diagnosing the problem is merely the first step: the operator must still find a fix that solves the problem, without causing new problems elsewhere. However, most existing debuggers focus exclusively on diagnosis and offer the network operator little or no help with finding an effective fix. Finding a suitable fix is difficult because the number of candidates can be enormous.

In this paper, we propose a step towards automated repair for SDN applications. Our approach consists of two elements. The first is a data structure that we call *meta provenance*, which can be used to efficiently find good candidate repairs. Meta provenance is inspired by the provenance concept from the database community; however, whereas standard provenance can only reason about changes to *data*, meta provenance can also reason about changes to *programs*. The second element is a system that can efficiently backtest a set of candidate repairs using historical data from the network. This is used to eliminate candidate repairs that do not work well, or that cause other problems.

We have implemented a system that maintains meta provenance for SDNs, as well as a prototype debugger that uses the meta provenance to automatically suggest repairs. Results from several case studies show that, for problems of moderate complexity, our debugger can find high-quality repairs within one minute.

1 Introduction

Debugging networks is notoriously hard. The advent of software-defined networking (SDN) has added a new dimension to the problem: networks can now be controlled by software programs, and, like all other programs, these programs can have bugs.

There is a substantial literature on network debugging and root cause analysis [16, 21, 23, 25, 36, 55, 61]. These tools can offer network operators a lot of help with debugging. For instance, systems like NetSight [21] and negative provenance [55] provide a kind of "backtrace" to capture historical executions, analogous to a stack trace in a conventional debugger, that can link an observed effect of a bug (say, packets being dropped in the network) to its root causes (say, an incorrect flow entry).

However, in practice, diagnosing the problem is only the first step. Once the root cause of a problem is known, the operator must find an effective fix that not only solves the problem at hand, but also avoids creating *new* problems elsewhere in the network. Given the complexity of modern controller programs and configuration files, finding a good fix can be as challenging as – or perhaps even more challenging than – diagnostics, and it often requires considerable expertise. However, current tools offer far less help with this second step than with the first.

In this paper, we present a step towards automated bug fixing in SDN applications. Ideally, we would like to provide a "Fix it!" button that automatically finds and fixes the root cause of an observed problem. However, removing the human operator from the loop entirely seems risky, since an automated tool cannot know the operator's intent. Therefore we opt for a slightly less ambitious goal, which is to provide the operator with a list of suggested repairs.

Our approach is to leverage and enhance some concepts that have been developed in the database community. For some time, this community has been studying the question how to explain the presence or absence of certain data tuples in the result of a database query, and whether and how the query can be adjusted to make certain tuples appear or disappear [9, 50]. By seeing SDN programs as "queries" that operate on a "database" of incoming packets and produce a "result" of delivered or dropped packets, it should be possible to ask similar queries – e.g., why a given packet was absent (misrouted/dropped) from an observed "result".

The key concept in this line of work is that of *data provenance* [6]. In essence, provenance tracks causality: the provenance of a tuple (or packet, or data item) consists of the tuples from which it was directly derived. By applying this idea recursively, it is possible to trace the provenance of a tuple in the output of a query all the way to the "base tuples" in the underlying databases. The result is a comprehensive causal explanation of how the tuple came to exist. This idea has previously been adapted for the SDN setting as *network provenance*, and it has been used, e.g., in debuggers and forensic tools such as ExSPAN [63], SNP [61] and Y! [55]. However, *so far this work has considered provenance only in terms of packets and configuration data – the SDN controller program was assumed to be immutable*. This is sufficient for diagnosis, but not for repair: we must also be able to infer which parts of the controller program were respon-

sible for an observed event, and how the event might be affected by changes to that program.

In this paper, we take the next step and extend network provenance to *both* programs *and* data. At a high level, we accomplish this with a combination of two ideas. First, we treat programs as just another kind of data; this allows us to reason about the provenance of data not only in terms of the data it was computed from, but also in terms of the parts of the program it was computed *with*. Second, we use counterfactual reasoning to enable a form of negative provenance [55], so that operators can ask why some condition did *not* hold (Example: "Why didn't any DNS requests arrive at the DNS server?"). This is a natural way to phrase a diagnostic query, and the resulting meta provenance is, in essence, a tree of changes (to the program and/or to configuration data) that could make the condition true.

Our approach presents three key challenges. First, there are infinitely many possible repairs to a given program (including, e.g., a complete rewrite), and not all of them will make the condition hold. To address this challenge, we show how to find suitable repairs efficiently using properties of the provenance itself. Second, even if we consider only suitable changes, there are still infinitely many possibilities. We leverage the fact that most bugs affect only a small part of the program, and that programmers tend to make certain errors more often than others [27, 41]. This allows us to rank the possible changes according to plausibility, and to explore only the most plausible ones. Finally, even a small change that fixes the problem at hand might still cause problems elsewhere in the network. To avoid such fixes, we backtest them using historical information that was collected in the network. In combination, this approach enables us to produce a list of suggested repairs that 1) are small and plausible, 2) fix the problem at hand, and 3) are unlikely to affect unrelated parts of the network.

We present a concrete algorithm that can generate meta provenance for arbitrary controller programs, as well as a prototype system that can collect the necessary data in SDNs and suggest repairs. We have applied our approach to three different controller languages, and we report results from several case studies; our results show that our system can generate high-quality repairs for realistic bugs, typically in less than one minute.

2 Overview

We illustrate the problem with a simple scenario (Figure 1). A network operator manages an SDN that connects two web servers and a DNS server to the Internet. To balance the load, incoming web requests are forwarded to different servers based on their source IP. At some point, the operator notices that web server H2 is not receiving any requests from the Internet.

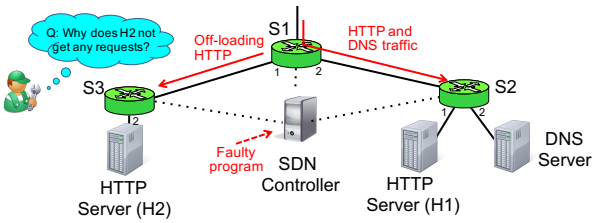


Figure 1: Example scenario. The primary web server (H1) is too busy, so the network offloads some traffic to a backup server (H2). The offloaded requests are never received because of a bug in the controller program.

Our goal is to build a debugger that accepts a simple specification of the observed problem (such as "H2 is not receiving any traffic on TCP port 80") and returns a) a detailed causal explanation of the problem, and b) a ranked list of suggested fixes. We consider a suggested fix to be useful if it a) fixes the specified problem and b) has few or no side-effects on the rest of the network.

2.1 Background: Network Datalog

Since our approach involves tracking causal dependencies, we will explain it using a declarative language, specifically *network datalog (NDlog)* [34], which makes these dependencies obvious. However, these dependencies are fundamental, and they exist in all the other languages that are used to program SDNs. To demonstrate this, we have applied our approach to three different languages, of which only one is declarative; for details, please see Section 5.8.

In NDlog, the state of a node (switch, controller, or server) is modeled as a set of *tables*, each of which contains a number of *tuples* (e.g., configuration state or network events). For instance, an SDN switch might contain a table called `FlowTable`, where each tuple represents a flow entry. Tuples can be manually inserted or programmatically derived from other tuples; the former are called *base tuples* and the latter *derived tuples*.

NDlog programs consist of *rules* that describe how tuples should be derived from each other. For example, the rule `A(@X,P):-B(@X,Q),Q=2*P` says that a tuple `A(@X,P)` should be derived on node `X` whenever a) there is also a tuple `B(@X,Q)` on that node, and b) `Q=2*P`. The `@` symbol specifies the node on which the tuple resides, and the `:-` symbol is the derivation operator. Rules may include tuples from different nodes; for instance, `C(@X,P):-C(@Y,P)` says that tuples in table `C` on node `Y` should be sent to node `X`.

2.2 Classical provenance

In NDlog, it is easy to see why a given tuple exists: if the tuple was derived using some rule `r` (e.g., `A(@X,5)`), then it must be the case that all the predicates in `r` were

```
r1 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), WebLoadBalancer(@C,Hdr,Prt), Swi == 1.
r2 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 1, Hdr == 53, Prt := 2.
r3 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 1, Hdr != 53, Prt := -1.
r4 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 1, Hdr != 80, Prt := -1.
r5 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 2, Hdr == 80, Prt := 1.
r6 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 2, Hdr == 53, Prt := 2.
r7 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 2, Hdr == 80, Prt := 2.
```

Figure 2: Part of an SDN controller program written in NDlog: Switch S1 load-balances HTTP requests across S2 and S3 (rule `r1`), forwards DNS requests to S3 (rule `r2`); and drops all other traffic (rules `r3`–`r4`). S2 and S3 forward the traffic to the correct server based on the destination port (rules `r5`–`r7`). The bug from Section 2.3 is underlined.

true (e.g., `B(@X,10)`), and all the constraints in `r` were satisfied (e.g., `10=2*5`.). This concept can be applied recursively (e.g., to explain the existence of `B(@X,10)`) until a set of base tuples is reached that cannot be explained further (e.g., configuration data or packets at border routers). The result is as a *provenance tree*, in which each vertex represents a tuple and edges represent direct causality; the root tuple is the one that is being explained, and the base tuples are the leaves. Using negative provenance [55], we can also explain why a tuple *does not* exist, by reasoning counterfactually about how the tuple *could* have been derived.

2.3 Case study: Faulty program

We now return to the scenario in Figure 1. One possible reason for this situation is that the operator has made a copy-and-paste error when writing the program. Figure 2 shows part of the (buggy) controller program: when the operator added the second web server H2, she had to update the rules for switch S3 to forward HTTP requests to H2. Perhaps she saw that rule `r5`, which is used for sending HTTP requests from S2 to H1, seemed to do something similar, so she copied it to another rule `r7` and changed the forwarding port, but forgot to change the condition `Swi==2` to check for S3 instead of S2.

When the operator notices that no requests are arriving at H2, she can use a provenance-based debugger to get a causal explanation. Provenance trees are more useful than large packet traces or the system-wide configuration files because they only contain information that is causally related to the observed problem. But the operator is still largely on her own when interpreting the provenance information and fixing the bug.

2.4 Meta provenance

Classical provenance is inherently unable to generate fixes because it reasons about the provenance of data that was generated *by a given program*. To find a fix, we also need the ability to reason about program changes.

We propose to add this capability, in essence, *by treating the program as just another kind of data*. Thus, the provenance of a tuple that was derived via a certain rule `r` does not only consist of the tuples that triggered `r`, but also of the syntactic components of `r` itself. For instance, when generating the provenance that explains why, in the scenario from Figure 1, no HTTP requests are arriving at H2, we eventually reach a point where we must explain the absence of a flow table entry in switch S3 that would send HTTP packets to port #2 on that switch. At this point, we can observe that rule `r7` would *almost* have generated such a flow entry, were it not for the predicate `Swi==2`, which did not hold. We can then, analogous to negative provenance, use counterfactual reasoning to determine that the rule *would* have the desired behavior if the constant were 3 instead of 2. Thus, the fact that the constant in the predicate is 2 and not 3 should become part of the missing flow entry's meta provenance.

2.5 Challenges

An obvious challenge with this approach is that there are infinitely many possible changes to a given program: constants, predicates, and entire rules can be changed, added, or deleted. However, in practice, only a tiny subset of these changes is actually relevant. Observe that, at any point in the provenance tree, we know exactly what we need to explain – e.g., the absence of a particular flow entry for HTTP traffic. Thus, we need not consider changes to the destination port in the header (`Hdr`) in `r7` (because that predicate is already true) or to unrelated rules that do not generate flow entries.

Of course, the number of relevant changes, and thus the size of any meta provenance graph, is still infinite. This does mean that we can never fully draw or materialize it – but there is also no need for that. Studies have shown that "real" bugs are often small [41], such as off-by-one errors or missing predicates. Thus, it seems useful to define a cost metric for changes (perhaps based on the number of syntactic elements they touch), and to explore only the "cheapest" changes.

Third, it is not always obvious what to change in order to achieve a desired effect. For instance, when changing `Swi==2` in the above example, why did we change the constant to 3 and not, say, 4? Fortunately, we can use existing tools, such as SMT solvers, that can enumerate possibilities quickly for the more difficult cases.

Finally, even if a change fixes the problem at hand, we cannot be sure that it will not cause new problems

```
program ← rule | rule program
rule    ← id func ":-" funcs "," sels "," assigns "."
id      ← (0-9a-zA-Z)+
funcs   ← func | func func
func    ← table "(" location "," arg "," arg ")"
table   ← (a-zA-Z)+
assigns ← assign | assign assigns
assign  ← arg ":=" expr
sels    ← sel "," sel
sel     ← expr opr expr
opr     ← == | < | > | !=
expr    ← integer | arg
```

Figure 3: μDlog grammar

elsewhere. Such side-effects are difficult to capture in the meta provenance itself, but we show that they can be estimated in another way, namely by backtesting changes with historical information from the network.

3 Meta Provenance

In this section, we show how to derive a simple meta provenance graph for both positive and negative events. We begin with a basic provenance graph for declarative programs, and then extend it to obtain meta provenance.

For ease of exposition, we explain our approach using a toy language, which we call μDlog. In essence, μDlog is a heavily simplified variant of NDlog: all tables have exactly two columns; all rules have one or two predicates and exactly two selection predicates, all selection predicates must use one of four operators (<, >, !=, ==), and there are no data types other than integers. The grammar of this simple language is shown in Figure 3. The controller program from our running example (in Figure 2) happens to already be a valid μDlog program.

3.1 The basic provenance graph

Recall from Section 2.2 that provenance can be represented as a DAG in which the vertices are events and the edges indicate direct causal relationships. Since NDlog's declarative syntax directly encodes dependencies, we can define relatively simple provenance graphs for it. For convenience, we adopt a graph from our prior work [55], which contains the following *positive* vertexes:

- EXIST$([t_1, t_2], N, \tau)$: Tuple τ existed on node N from time t_1 to t_2;
- INSERT(t, N, τ), DELETE(t, N, τ): Base tuple τ was inserted (deleted) on node N at time t;
- DERIVE(t, N, τ), UNDERIVE(t, N, τ): Derived tuple τ acquired (lost) support on N at time t;
- APPEAR(t, N, τ), DISAPPEAR(t, N, τ): Tuple τ appeared (disappeared) on node N at time t; and
- SEND$(t, N \rightarrow N', \pm\tau)$, RECEIVE$(t, N \leftarrow N', \pm\tau)$: $\pm\tau$ was sent (received) by node N to/from N' at t.

Conceptually, the system builds the provenance graph incrementally at runtime: whenever a new base tuple is inserted, the system adds an INSERT vertex, and whenever a rule is triggered and generates a new derived tuple, the system adds a DERIVE vertex. The APPEAR and EXIST vertexes are generated whenever a tuple is added to the database (after an insertion or derivation), and the interval in the EXIST vertex is updated once the tuple is deleted again. The rules for DELETE, UNDERIVE, and DISAPPEAR are analogous. The SEND and RECEIVE vertexes are used when a rule on one node has a tuple τ on another node as a precondition; in this case, the system sends a message from the latter to the former whenever τ appears (+τ) or disappears (-τ), and the two vertexes are generated when this message is sent and received, respectively. Notice that – at least conceptually – vertexes are never deleted; thus, the operator can inspect the provenance of past events.

The system inserts an edge (v_1, v_2) between two vertexes v_1 and v_2 whenever the event represented by v_1 is a direct cause of the event represented by v_2. Derivations are caused by the appearance (if local) or reception (if remote) of the tuple that satisfies the last precondition of the corresponding rule, as well as by the existence of any other tuples that appear in preconditions; appearances are caused by derivations or insertions, message transmissions by appearances, and message arrivals by message transmissions. The rules for underivations and disappearances are analogous. Base tuple insertions are external events that have no cause within the system.

So far, we have described only the vertexes for positive provenance. The full graph also supports *negative* events [55] by introducing a negative "twin" for each vertex. For instance, the counterpart to APPEAR is NAPPEAR, which represents the fact that a certain tuple *failed* to appear. For a more detailed discussion of negative provenance, please see [55].

3.2 The meta provenance graph

The above provenance graph can only represent causality between data. We now extend the graph to track provenance of programs by introducing two elements: *meta tuples*, which represent the syntactic elements of the program itself (such as conditions and predicates) and *meta rules*, which describe the operational semantics of the language. For clarity, we describe the meta model for μDlog here; our meta model for the full NDlog language is more complex but follows the same approach.

Meta tuples: We distinguish between two kinds of meta tuples: program-based tuples and runtime-based tuples. Program-based tuples are the syntactic elements that are visible to the programmer: rule heads (`HeadFunc`), predicates (`PredFunc`), assignments (`Assign`), constants (`Const`), and operators (`Oper`). Runtime-based tuples describe data structures inside the NDlog runtime: base tuple insertions (`Base`), tuples (`Tuple`), sat-

```
h1 Tuple(@C,Tab,Val1,Val2) :- Base(@C,Tab,Val1,Val2).
h2 Tuple(@L,Tab,Val1,Val2) :- HeadFunc(@C,Rul,Tab,Loc,Arg1,Arg2), HeadVal(@C,Rul,JID,Loc,L), Val == True,
      HeadVal(@C,Rul,JID1,Arg1,Val1), HeadVal(@C,Rul,JID2,Arg2,Val2), Sel(@C,Rul,JID,SID,Val), Val' == True,
      Sel(@C,Rul,JID,SID',Val'), True == f_match(JID1,JID), True == f_match(JID2,JID), SID != SID'.
p1 TuplePred(@C,Rul,Tab,Arg1,Arg2,Val1,Val2) :- Tuple(@C,Tab,Val1,Val2), PredFunc(@C,Rul,Tab,Arg1,Arg2).
p2 PredFuncCount(@C,Rul,Count<N>) :- PredFunc(@C,Rul,Tab,Arg1,Arg2).
j1 Join4(@C,Rul,JID,Arg1,Arg2,Arg3,Arg4,Val1,Val2,Val3,Val4) :- TuplePred(@C,Rul,Tab,Arg1,Arg2,Val1,Val2),
      TuplePred(@C,Rul,Tab',Arg3,Arg4,Val3,Val4), PredFuncCount(@C,Rul,N), N==2, Tab != Tab', JID := f_unique().
j2 Join2(@C,Rul,JID,Arg1,Arg2,Val1,Val2) :- TuplePred(@C,Rul,Tab,Arg1,Arg2,Val1,Val2), PredFuncCount(@C,Rul,N),
      N == 1, JID := f_unique().
e1 Expr(@C,Rul,JID,ID,Val) :- Const(@C,Rul,ID,Val), JID := *.
e2 Expr(@C,Rul,JID,Arg1,Val1) :- Join2(@C,Rul,JID,Arg1,Arg2,Val1,Val2).
e3-e7 // analogous to e2 for Arg2/Val2 (Join2) and Arg1..4/Val1..4 (Join4)
a1 HeadVal(@C,Rul,JID,Arg,Val) :- Assign(@C,Rul,Arg,ID), Expr(@C,Rul,JID,ID,Val).
s1 Sel(@C,Rul,JID,SID,Val) :- Oper(@C,Rul,SID,ID',ID'',Opr), Expr(@C,Rul,JID',ID',Val'),
      Expr(@C,Rul,JID'',ID'',Val''), True == f_match(JID',JID''), JID := f_join(JID',JID''),
      Val := (Val' Opr Val''), ID' != ID''.
```

Figure 4: Meta rules for μDlog.

isfied predicates (`TuplePred`), evaluated expressions (`Expr`), joins (`Join`), selections (`Sel`) and values in rule heads (`HeadVal`). Although concrete implementations may maintain additional data structures (e.g., for optimizations), these tuples are sufficient to describe the operational semantics.

Meta rules: Figure 4 shows the full set of meta rules for μDlog. Notice that these rules are written in NDlog, not in μDlog itself. We briefly explain each meta rule below.

Tuples can exist for two reasons: they can be inserted as base tuples (`h1`) or derived via rules (`h2`). Recall that, in μDlog's simplified syntax, each rule joins at most two tables and has exactly two selection predicates to select tuples from these tables. A rule "fires" and produces a tuple `T(a,b)` iff there is an assignment of values to a, and b that satisfies both predicates. (Notice that the two selection predicates are distinguished by a unique selection ID, or `SID`.) We will return to this rule again shortly.

The next four meta rules compute the actual joins. First, whenever a (syntactic) tuple appears as in a rule definition, each concrete tuple that exists at runtime generates one variable assignment for that tuple (`p1`). For instance, if a rule `r` contains `Foo(A,B)`, where `A` and `B` are variables, and at runtime there is a concrete tuple `Foo(5,7)`, meta rule `p1` would generate a `TuplePred(@C,r,Foo,A,B,5,7)` meta tuple to indicate that `5` and `7` are assignments for `A` and `B`.

Depending on the number of tuples in the rule body (calcuated in rule `p2`), meta rule `j1` or `j2` will be triggered: When it contains two tuples from different tables, meta rule `j1` computes a `Join4` tuple for each pair of tuples from these tables. Note that this is a full cross-product, from which another meta rule (`s1`) will then select the tuples that match the selection predicates in the rule. For this purpose, each tuple in the join is given a unique join ID (`JID`), so that the values of the selection predicates can later be matched up with the correct tuples. If a rule contains only a tuple from one table, we compute a `Join2` tuple instead (`j2`).

The next seven meta rules evaluate expressions. Expressions can appear in two different places – in a rule head and in a selection predicate – but since the evaluation logic is the same, we use a single set of meta rules for both cases. Values can come from integer constants (`e1`) or from any element of a `Join2` or `Join4` meta tuple (`e2`–`e7`). Notice that most of these values are specific to the join on which they were evaluated, so each `Expr` tuple contains a specific `JID`; the only exception are the constants, which are valid for all joins. To capture this, we use a special JID wildcard ($\star$), and we test for JID equality using a special function `f_match(JID1,JID2)` that returns true iff `JID1==JID2` or if either of them is $\star$.

The last two meta rules handle assignments (`a1`) and selections (`s1`). An assignment simply sets a variable in a rule head to the value of an expression. The `s1` rule determines, for each selection predicate in a rule (identified by `SID`) and for each join state (identified by `JID`) whether the check succeeds or fails. Function `f_join(JID1, JID2)` is introduced to handle JID wildcard: it returns `JID1` if `JID2` is $\star$, or `JID2` otherwise. The result is recorded in a `Sel` meta tuple, which is used in `h2` to decide whether a head tuple is derived.

μDlog requires only 13 meta tuples and 15 meta rules; the full meta model for NDlog contains 23 meta tuples and 23 meta rules. We omit the details here; they are included in a technical report [54].

3.3 Meta provenance forests

So far, we have essentially transformed the original NDlog program into a new "meta program". In principle, we could now generate meta provenance graphs by applying a normal provenance graph generation algorithm on the meta program – e.g., the one from [55]. However, this is not quite sufficient for our purposes. The reason is that there are cases where the same effect can be achieved in multiple ways. For instance, suppose that we are explaining the absence of an `X` tuple, and that there are two different rules, `r1` and `r2`, that could derive `X`. If our goal

was to *explain why* X was absent, we would need to include explanations for both r1's and r2's failure to fire. However, our goal is instead to *make X appear*, which can be achieved by causing *either* r1 *or* r2 to fire. If we included both in the provenance tree, we would generate only repairs that cause both rules to fire, which is unnecessary and sometimes even impossible.

Our solution is to replace the meta provenance tree with a meta provenance *forest*. Whenever our algorithm encounters a situation with k possible choices that are each individually sufficient for repair, it replaces the current tree with k copies of itself and continues to explore only one choice in each tree.

3.4 From explanations to repairs

The above problem occurs in the context of disjunctions; next, we consider its "twin", which occurs in the context of conjunctions. Sometimes, the meta provenance must explain why a rule with multiple preconditions did *not* derive a certain tuple. For diagnostic purposes, the absence of one missing precondition is already sufficient to explain the absence of the tuple. However, meta provenance is intended for repair, i.e., it must allow us to find a way to make the missing tuple appear. Thus, it is not enough to find a way to make a single precondition true, or even ways to make each precondition true individually. What we need is a way to satisfy *all* the preconditions *at the same time*!

For concreteness, consider the following simple example, which involves a meta rule `A(x,y):-B(x), C(x,y),x+y>1,x>0`. Suppose that the operator would like to find out why there is no `A(x,y)` with `y==2`. In this case, it would be sufficient to show that there is no `C(x,y)` with `y==2` and `x>0`; cross-predicate constraints, such as `x+y>1`, can be ignored. However, if we want to actually make a suitable `A(x,y)` appear, we need to *jointly* consider the absence of both `B(x)` and `C(x,y)`, and ensure that all branches of the provenance tree respect the cross-predicate constraints. In other words, we cannot explore the two branches separately; we must make sure that their contents "match".

To accomplish this, our algorithm automatically generates a constraint pool for each tree. It encodes the attributes of tuples as variables, and it formulates constraints over these variables. For instance, given the missing tuple A_0, we add two variables A_0`.x` and A_0`.y`. To initialize the constraint pool, the root of the meta provenance graph must satisfy the operator's requirement: A_0`.y == 2`. While expanding any missing tuple, the algorithm adds constraints as necessary for a successful derivation. In this example, three constraints are needed: first, the predicates must join together, i.e., B_0`.x ==` C_0`.x`. Second, the predicates must satisfy the constraints, i.e., B_0`.x>0` and C_0`.x+`C_0`.y>1`. Third, the predicates must derive the head, i.e., A_0`.x==`C_0`.x` and A_0`.y==`C_0`.y`. In addition, tuples must satisfy primary key constraints. For instance, suppose deriving `B(x)` requires D_0`(9,1)` while deriving `C(x,y)` requires D_1`(9,2)`. If `x` is the only primary key of table `D(x,y)`, D_0`(9,1)` and D_1`(9,2)` cannot co-exist at the same time. Therefore, the explanation is inconsistent for generating repairs. To address such cases, we encode additional constraints: `D.x ==` D_0`.x implies D.y == 1` and `D.x ==` D_1`.x implies D.y == 2`.

3.5 Generating meta provenance

In general, meta provenance forests may consist of infinitely many trees, each with infinitely many vertexes. Thus, we cannot hope to materialize the entire forest. Instead, we adopt a variant of the approach from [55] and use a step-by-step procedure that constructs the trees incrementally. We define a function QUERY(v) that, when called on a vertex v from any (partial) tree in the meta provenance forest, returns the immediate children of v and/or "forks" the tree as described above. By calling this function repeatedly on the leaves of the trees, we can explore the trees incrementally. The two key differences to [55] are the procedures for expanding NAPPEAR and NDERIVE vertices: the former must now "fork" the tree when there are multiple children that are each individually sufficient to make the missing tuple appear (Section 3.3), and the latter must now explore a join across *all* preconditions of a missing derivation, while collecting any relevant constraints (Section 3.4).

To explore an infinite forest with finite memory, our algorithm maintains a set of partial trees. Initially, this set contains a single "tree" that consists of just one vertex – the vertex that describes the symptom that the operator has observed. Then, in each step, the algorithm picks one of the partial trees, randomly picks a vertex within that tree that does not have any children yet, and then invokes QUERY on this vertex to find the children, which are then added to that tree. As discussed before, this step can cause the tree to fork, adding multiple copies to the set that differ only in the newly added children. Another possible outcome is that the chosen partial tree is completed, which yields a repair candidate.

Each tree – completed or partial – is associated with a *cost*, which intuitively represents the implausibility of the repair that the tree represents. (Lower-cost trees are more plausible.) Initially, the cost is zero. Whenever a base tuple is added that represents a program change, we increase the total cost of the corresponding tree by the cost of that change. In each step, our algorithm picks the partial tree with the lowest cost; if there are multiple trees with the same cost, our algorithm picks the one

```
function GenerateRepairCandidates(P)
    R ← ∅, τ_r ← RootTuple(P)
    if MissingTuple(τ_r) then
        C ← ConstraintPool(P)
        A ← SatAssignment(C)
        for (τ_i) ∈ BaseTuples(P)
            if MissingTuple(τ_i) then
                R ← R ∪ ChangeTuple(τ_i,A)
    else if ExistingTuple(τ_r) then
        for (T_i) ∈ BaseTupleCombinations(P)
            R_ci ← ∅, R_di ← ∅
            C_i ← SymbolicPropagate(P,T_i)
            A_i ← UnsatAssignment(C_i)
            for (τ_i) ∈ T_i
                R_ci ← R_ci ∪ ChangeTuple(τ_i,A_i)
                R_di ← R_di ∪ DeleteTuple(τ_i)
            R ← R ∪ R_ci ∪ R_di
    return R
```

Figure 5: Algorithm for extracting repair candidates from the meta provenance graph. For a description of the helper functions, please see [54].

with the smallest number of unexpanded vertexes. Repair candidates are output only once there are no trees with a lower cost. Thus, repair candidates are found in cost order, and the first one is optimal with respect to the chosen cost metric; if the algorithm runs long enough, it should eventually find a working repair. (For a more detailed discussion, please see [54].) In practice, the algorithm would be run until some reasonable cut-off cost is reached, or until the operator's patience runs out.

The question remains how to assign costs to program changes. We assign a low cost to common errors (such as changing a constant by one or changing a == to a !=) and a high cost to unlikely errors (such as writing an entirely new rule, or defining a new table). Thus, we can prioritize the search of fixes to software bugs that are more commonly observed in actual programming, and thus increase the chances that a working fix will be found.

3.6 Limitations

The above approach is likely to find simple problems, such as incorrect constraints or copy-and-paste errors, but it is not likely to discover fundamental flaws in the program logic that require repairs in many different places and/or several new rules. However, software engineering studies have consistently shown that simple errors, such as copy-and-paste bugs, are very common: simple typos already account for 9.4-9.8% of all semantic bugs [32], and 70–90% of bugs can be fixed by changing only existing syntactic elements [41]. Because of this, we believe that an approach that can automatically fix "low-cost" bugs can still be useful in practice.

Our approach focuses exclusively on incorrect computations; there are classes of bugs, such as concurrency bugs or performance bugs, that it cannot repair. We speculate that such bugs can be found with a richer meta model, but this is beyond the scope of the present paper.

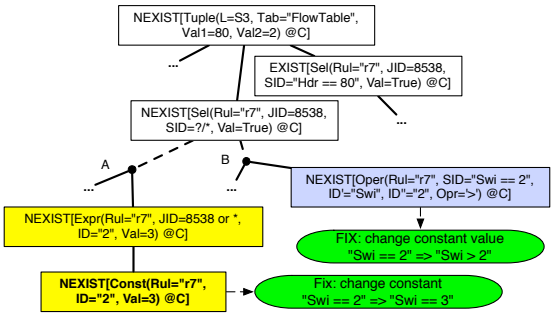


Figure 6: Meta provenance of a missing flow entry. It consists of two trees (white + yellow, white + blue), each of which can generate a repair candidate.

4 Generating repair candidates

As discussed in Section 3.5, our algorithm explores the meta provenance forest in cost order, adding vertexes one by one by invoking QUERY on a leaf of an existing partial tree. Thus, the algorithm slowly generates more and more trees; at the same time, some existing trees are eventually completed because none of their leaves can be further expanded (i.e., QUERY returns $\emptyset$ on them). Once a tree is completed, we invoke the algorithm in Figure 5 to extract a candidate repair.

The algorithm has two cases: one for trees that have an existing tuple at the root (e.g., a packet that reached a host it should not have reached), and one for trees that have a missing tuple at the root (e.g., a packet failed to reach its destination). We discuss each in turn. Furthermore, we note that the ensuing analysis is performed on the *meta* program, which is independent from the language that the *original* program is written in.

4.1 Handling negative symptoms

If the root of the tree is a missing tuple, its leaves will contain either missing tuples or missing meta tuples, which can be then created by inserting the corresponding tuples or program elements. However, some of these tuples may still contain variables – for instance, the tree might indicate that an `A(x)` tuple is missing, but without a concrete value for `x`. Hence, the algorithm first looks for a satisfying assignment of the tree's constraint pool (Section 3.4). If such an assignment is found, it will supply concrete values for all remaining variables; if not, the tree cannot produce a working repair and is discarded.

As an example, Figure 6 shows part of the meta provenance of a missing event. It contains two meta provenance trees, which have some vertices in common (colored white), but do not share other vertices (colored yellow and blue). The constraint pool includes `Const0.Val = 3`, `Const0.Rul = r7`, and `Const0.ID = 2`. That is, the repair requires the exis-

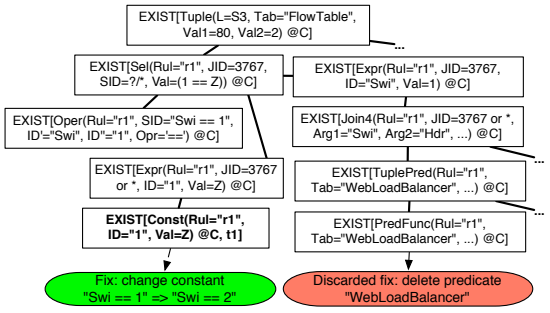


Figure 7: Meta provenance of a harmful flow entry. All repairs (e.g., green and red) can prevent this derivation, but the red one rederives the tuple via other meta rules.

tence of a constant of value `3` in rule `r7`. Therefore, we can change value of the original constant (identified by identical primary keys `Rul` and `ID`) to `3`.

4.2 Handling positive symptoms

Meta provenance can also help with debugging scenarios with positive symptoms. Figure 7 shows the meta provenance graph of a tuple that exists, but should not exist. We can make this tuple disappear by deleting (or changing in the proper way) any of the base tuples or meta tuples on which the derivation is based.

However, neither base tuples nor meta tuples are always safe to change. In the case of meta tuples, we must ensure that the change does not violate the syntax of the underlying language (in this case, μDlog). For instance, it would be safe to delete a `PredFunc` tuple to remove an entire predicate, but it may not be safe to delete a `Const` meta tuple, since this might result in an incomplete expression, such as `Swi >`.

In the case of changes to base tuples, the problem is to find changes that a) will make the current derivation disappear, and that b) will *not* cause an alternate derivation of the same tuple via different meta rules. To handle the first problem, we do not directly replace elements of a tuple with a different value. Rather, we initially replace the elements with symbolic constants and then re-execute the derivation of meta rules symbolically while collecting constraints over the symbolic constants that must hold for the derivation to happen. Finally, we can negate these constraints and use a constraint solver to find a satisfying assignment for the negation. If successful, this will yield concrete values we can substitute for the symbolic constant that will make the derivation disappear.

For concreteness, we consider the green repair in Figure 7. We initially replace `Const('r1',1,1)` with `Const('r1',1,Z)` and then reexecute the derivation to collect constraints – in this case, `1==Z`. Since `Z=2` does not satisfy the constraints, we can make the tuple at the top disappear by changing Z to 2 (which corresponds to changing `Swi==1` to `Swi==2` in the program).

This leaves the second problem from above: even if we make a change that disables one particular derivation of an undesired tuple, that very change could enable some other derivation that causes the undesired tuple to reappear. For instance, suppose we delete the tuple `PredFunc('r1','WebLoadBalancer', ...)`, which corresponds to deleting the `WebLoadBalancer` predicate from the μDlog rule `r1` (shaded red in Figure 7). This deletion will cause the `Join4` tuple to disappear, and it will change the value of `PredFuncCount` from 2 to 1. As a result, the derivation through meta rule `j1` will duly disappear; however, this will instead trigger meta rule `j2`, which leads to another derivation of the same flow entry.

Solving this for arbitrary programs is equivalent to solving the halting problem, which is NP-hard. However, we do not need a perfect solution because this case is rare, and because we can either use heuristics to track certain rederivations or we can easily eliminate the corresponding repair candidates during backtesting.

4.3 Backtesting a single repair candidate

Although the generated repairs will (usually) solve the problem immediately at hand, by making the desired tuple appear or the undesired tuple disappear, each repair can also have a broader effect on the network as a whole. For instance, if the problem is that a switch forwarded a packet to the wrong host, one possible "repair" is to disable the rule that generates flow entries for that switch. However, this would also prevent *all other* packets from being forwarded, which is probably too restrictive.

To mitigate this, we adopt the maxim of "primum non nocere" [20] and assess the global impact of a repair candidate before suggesting it. Specifically, we backtest the repair candidates in simulation, using historical information from the network. We can approximate past control-plane states from the diagnostic information we already record for the provenance; to generate a plausible workload, we can use a Netflow trace or a sample of packets. We then collect some key statistics, such as the number of packets delivered to each host. Since the problems we are aiming to repair are typically subtle (total network failures are comparatively easy to diagnose!), they should affect only a small fraction of the traffic. Hence, a "good" candidate repair should have little or no impact on metrics that are not related to the specified problem.

In essence, the metrics play the role of the test suite that is commonly used in the wider literature on automated program fixing. While the simple metric from above should serve as a good starting point, operators could easily add metrics of their own, e.g., to encode

```
r7(v1) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 3, Hdr == 80, Prt := 2.
```

```
r7(v2) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi > 2, Hdr == 80, Prt := 2.
```

```
r7(v3) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi != 2, Hdr == 80, Prt := 2.
```

(a)

```
r6(v1,v2,v3) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 2, Hdr == 53, Prt := 2.
r7(v1,v2,v3) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 3, Hdr == 80, Prt := 2.
r7(v2,v3) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi > 3, Hdr == 80, Prt := 2.
r7(v3) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi < 2, Hdr == 80, Prt := 2.
```

(b)

Figure 8: (a) Three repair candidates, all of which can generate forwarding flow entries for switch S2 by fixing `r7` in the original program in Figure 2; other parts of the program are unchanged. (b) Backtesting program that evaluates all three repair candidates simultaneously while running shared computations only once.

performance goals (load balancing, link utilization) or security restrictions (traffic from X should never reach Y). However, recall that, in contrast to much of the earlier work on program fixing, we do *not* rely on this "test suite" to *find* candidate repairs (we use the meta provenance for that); the metrics simply serve as a sanity check to weed out repairs with serious side effects. The fact that a given repair passed the backtesting stage is not a guarantee that no side effects will occur.

As an additional benefit, the metrics can be used to rank the repairs, and to give preference to the candidates that have the smallest impact on the overall network.

4.4 Backtesting multiple repair candidates

It is important for the backtesting to be fast: the less time it takes, the more candidate repairs we can afford to consider. Fortunately, we can leverage another concept from the database literature to speed up this process considerably. Recall that each backtest simulates the behavior of the network with the repaired program. Thus, we are effectively running many very similar "queries" (the repaired programs, which differ only in the fixes that were applied) over the same "database" (the historical network data), where we expect significant overlaps among the query computations. This is a classical instance of *multi-query optimization*, for which powerful solutions are available in the literature [19, 35].

Multi-query optimization exploits the fact that almost all computation is shared by almost all repair candidates, and thus has to be performed only once. We accomplish this by transforming the original program into a *backtesting program* as follows. First, we associate each tuple with a set of tags, we extend all relations to have a new field for storing the tags, and we update all the rules such that the tag of the head is the intersection of the tags in the body. Then, for each repair candidate, we create a new tag and add copies of all the rules the repair candidate modifies, but we restrict them to this particular tag. Finally, we add rules that evaluate the metrics from Section 4.3, separately for each tag.

The effect is that data flows through the program as usual, but, at each point where a repair candidate has modified something, the flow forks off a subflow that has the tag of that particular candidate. Thus, the later in the program the modification occurs, the fewer computations have to be duplicated for that candidate. Overall, the backtesting program correctly computes the metrics for each candidate, but runs considerably faster than computing each of the metrics round after round.

As an example, Figure 8(a) shows three repair candidates (`v1`, `v2`, and `v3`) for the buggy program in Figure 2. Each of them alters the rule `r7` in a different way: `v1` changes a constant, `v2` and `v3` change an operator. (Other rules are unchanged.)

In some cases, it is possible to determine, through static analysis, that rules with different tags produce overlapping output. For instance, in the above example, the three repairs all modify the same predicate, and some of the predicates are implied by others; thus, the output for switch 3 is the same for all three tags, and the output for switches above 3 is the same for tags `v2` and `v3`. By coalescing the corresponding rules, we can further reduce the computation cost. Finding *all* opportunities for coalescing would be difficult, but recall that this is merely an optimization: even if we find none at all, the program will still be correct, albeit somewhat slower.

5 Evaluation

In this section, we report results from our experimental evaluation, which aim to answer five high-level questions: 1) Can meta provenance generate reasonable repair candidates? 2) What is the runtime overhead of meta provenance? 3) How fast can we process diagnostic queries? 4) Does meta provenance scale well with the network size? And 5) how well does meta provenance work across different SDN frameworks?

5.1 Prototype implementation

We have built a prototype based on declarative and imperative SDN environments as well as Mininet [29]. It

generates and further backtests repair candidates, such that the operator can inspect the suggested repairs and decide whether and which to apply. Our prototype consists of around 30,000 lines of code, including the following three main components.

Controllers: We validate meta provenance using three types of SDN environments. The first is a declarative controller based on RapidNet [44]; it includes a proxy that interposes between the RapidNet engine and the Mininet network and that translates NDlog tuples into OpenFlow messages and vice versa. The other two are existing environments: the Trema framework [51] and the Pyretic language [37]. (Notice that neither of the latter two is declarative: Trema is based on Ruby, an imperative language, and Pyretic is an imperative domain-specific language that is embedded in Python.)

At runtime, the controller and the network each record relevant control-plane messages and packets to a log, which can be used to answer diagnostic queries later. The information we require from runtime is not substantially different from existing provenance systems [10, 33, 55, 63], which have shown that provenance can be captured at scale and for SDNs.

Tuple generators: For each of the above languages, we have built a meta tuple generator that automatically generates meta tuples from the controller program and from the log. The program-based meta tuples (e.g., constants, operators, edges) only need to be generated once for each program; the log-based meta tuples (e.g., messages, constraints, expressions) are generated by replaying the logged control-plane messages through automatically-instrumented controller programs.

Tree constructor: This component constructs meta provenance trees from the meta tuples upon a query. As we discussed in Section 3.4, this requires checking the consistency of repair candidates. Our constructor has an interface to the Z3 solver [11] for this purpose. However, since many of the constraint sets we generate are trivial, we have built our own "mini-solver" that can quickly solve the trivial instances on its own; the nontrivial ones are handed over to Z3. The mini-solver also serves as an optimizer for handling cross-table meta tuple joins. Using a naïve nested loop join that considers all combinations of different meta tuples would be inefficient; instead, we solve simple constraints (e.g., equivalence, ranges) first. This allows us to filter the meta tuples before joining them, and use more efficient join paradigms, such as hash joins. Our cost metric is based on a study of common bug fix patters (Pan et al. [41]).

5.2 Experimental setup

To obtain a representative experimental environment, we set up the Stanford campus network from ATPG [58] in Mininet [29], with 16 Operational Zone and backbone routers. Moreover, we augmented the topology with edge networks, each of which is connected to the main network by at least one core router; we also set up 1 to 15 end hosts per edge network. The core network is proactively configured using forwarding entries from the Stanford campus network; the edge networks run a mix of reactive and proactive applications. In our techinical report [54], we include an experiment where the controller reactively installs core routing policies. Overall, our smallest topology across all scenarios consisted of 19 routers and 259 hosts, and our largest topology consisted of 169 routers and 549 hosts. In addition, we created realistic background traffic using two traffic traces obtained in a similar campus network setting [5]; 1 to 16 of the end hosts replayed the traces continuously during the course of our experiments. Moreover, we generated a mix of ICMP `ping` traffic and HTTP web traffic on the remaining hosts. Overall, 4.6–309.4 million packets were sent through the network. We ran our experiments on a Dell OptiPlex 9020 workstation, which has a 8-core 3.40 GHz Intel i7-4770 CPU with 16 GB of RAM and a 128 GB OCZ Vector SSD. The OS was Ubuntu 13.10, and the kernel version was 3.8.0.

5.3 Usability: Diagnosing SDNs

A natural first question to ask is whether meta provenance can repair real problems. To avoid distorting our results by picking our own toy problems to debug, we have chosen four diagnostic scenarios from four different networking papers that have appeared at CoNEXT [13, 58], NSDI [7], and HotSDN [4], plus one common class of bugs from an OSDI paper [31]. We focused on scenarios where the root cause of the problem was a bug in the controller program. We recreated each scenario in the lab, based on its published description. The five scenarios were:

- **Q1: Copy-and-paste error [31].** A server received no requests because the operator made a copy-and-paste error when modifying the controller program. The scenario is analogous to the one in Figure 1, but with larger topology and more realistic traffic.
- **Q2: Forwarding error [58].** A server could not receive queries from certain clients because the operator made a error when specifying the action of the forwarding rule.
- **Q3: Uncoordinated policy update [13].** A firewall controller app configured white-list rules for web servers. A load-balancing controller app updated the policy on an ingress point, without coordinating with the firewall app; this caused some traffic to shift, and then to be blocked by the firewall.

	Query description	Result
Q1	H20 is not receiving HTTP requests from H2	9/2
Q2	H17 is not receiving DNS queries from H1	12/3
Q3	H20 is not receiving HTTP requests from H1	11/3
Q4	First HTTP packet from H2 to H20 is not received	13/3
Q5	H2's MAC address is not learned by the controller	9/3

Table 1: The diagnostic queries, the number of repair candidates generated by meta provenance, and the number of remaining candidates after backtesting.

- **Q4: Forgotten packets [7].** A controller app correctly installed flow entries in response to new flows; however, it forgot to instruct the switches to forward the first incoming packet in each flow.
- **Q5: Incorrect MAC learning [4].** A MAC learning app should have matched packets based on their source IP, incoming port, and destination IP; however, the program only matched on the latter two fields. As a result, some switches never learned about the existence of certain hosts.

To get a sense of how useful meta provenance would be for repairing the problems, we ran diagnostic queries in our five scenarios as shown in Table 1, and examined the generated candidate repairs. In each of the scenarios, we bounded the cost and asked the repair generator to produce all repair candidates. Table 2 shows the repair candidates returned for Q1; the others are included in our technical report [54].

Our backtesting confirmed that each of the proposed candidates was effective, in the sense that it caused the backup web server to receive at least some HTTP traffic. This phase also weeded out the candidates that caused problems for the rest of the network. To quantify the side effects, we replayed historical packets in the original network and in each repaired network. We then computed the traffic distribution at end hosts for each of these networks. We used the Two-Sample Kolmogorov-Smirnov test with significance level 0.05 to compare the distributions before and after each repair. A repair candidate was rejected if it significantly distorted the original traffic distribution; the statistics and the decisions are shown in Table 2. For instance, repair candidate G deleted `Swi==2` and `Dpt==53` in rule `r6`. This causes the controller to generate a flow entry that forwards HTTP requests at S3; however, the modified `r6` *also* causes HTTP requests to be forwarded to the DNS server.

After backtesting, the remaining candidates are presented to the operator in complexity order, i.e., the simplest candidate is shown first. In this example, the second candidate on the list (B) is also the one that most human operators would intuitively have chosen – it fixes the copy-and-paste bug by changing the switch ID in the faulty predicate from `Swi==2` to `Swi==3`.

Table 1 summarizes the quality of repairs our prototype generated for all scenarios for the RapidNet controller. Each scenario resulted in two or three repair suggestions. In the first stage, meta provenance produced between 9 and 13 repair candidates for each query, for a total of 54 repair candidates. Note that these numbers do not count expensive repair candidates that were discarded by the ranking heuristic (Section 3.5). The backtesting stage then confirmed that 48 of these candidates were effective, i.e., they fixed the problem at hand (e.g., the repair caused the server to receive at least a few packets). However, 34 of the effective candidates caused non-trivial side effects, and thus were discarded.

	Repair candidate (Accepted?)	KS-test
A	Manually installing a flow entry (✓)	0.00007
B	Changing `Swi==2` in `r7` to `Swi==3` (✓)	0.00007
C	Changing `Swi==2` in `r7` to `Swi!=2` (✗)	0.00865
D	Changing `Swi==2` in `r7` to `Swi>=2` (✗)	0.00826
E	Changing `Swi==2` in `r7` to `Swi>2` (✗)	0.00826
F	Deleting `Swi==2` in `r7` (✗)	0.00867
G	Deleting `Swi==2` and `Dpt==53` in `r6` (✗)	0.05287
H	Deleting `Swi==2` and `Dpt==80` in `r7` (✗)	0.00999
I	Changing `Swi==2` and `Act=output-1` in `r5` to `Swi==3` and `Act=output-2` (✗)	0.05286

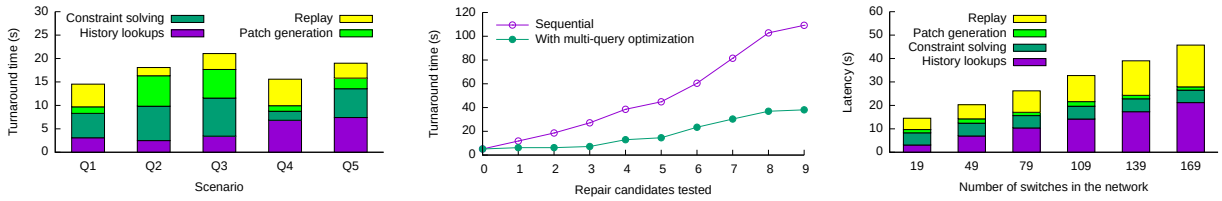
Table 2: Candidate repairs generated by meta provenance for Q1, which are then filtered by a KS-test.

We note that the final set of candidates included a few non-intuitive repairs – for instance, one candidate fixed the problem in Q1 by manually installing a new flow entry. However, these repairs were nevertheless effective and had few side effects, so they should suffice as an initial fix. If desired, a human operator could always refactor the program later on.

5.4 Runtime overhead

Latency and throughput: To measure the latency and throughput overhead incurred by maintaining meta provenance, we used a standard approach of stress-testing OpenFlow controllers [14] which involves streaming incoming packets through the Trema controller using `Cbench`. Latency is defined as the time taken to process each packet within the controller. We observe that provenance maintenance resulted in a latency increase of 4.2% to 54ms, and a throughput reduction of 9.8% to $45,423$ packets per second.

Disk storage: To evaluate the storage overhead, we streamed the two traffic traces obtained from [5] through our SDN scenario in Q1. For each packet in the trace, we recorded a 120-byte log entry that contains the packet header and the timestamp. The logging rates for the two traces are 20.2 MB/s and 11.4 MB/s per switch, respectively, which are only a fraction of the sequential write rate of commodity SSDs. Note that this data need not be kept forever: most diagnostic queries are about problems that currently exist or have appeared recently. Thus, is should be sufficient to store the most recent entries, perhaps an hour's worth.



(a) Time to generate the repairs for each of the scenarios in Section 5.3.

(b) Time needed to *jointly* backtest the first k repair candidates from Q1.

(c) Scalability of repair generation phase with network size for Q1.

Figure 9: Repair generation speed for all queries; backtesting speed and scalability result for Q1.

5.5 Time to generate repairs

Diagnostic queries does not always demand a real-time response; however, operators would presumably prefer a quick turnaround. Figure 9a shows the turnaround time for constructing the meta provenance data structure and for generating repair candidates, including a breakdown by category. In general, scenarios with more complex control-plane state (Q1, Q4, and Q5) required more time to query the time index and to look up historical data; the latter can involve loop-joining multiple meta tables, particularly for the more complicated meta rules with over ten predicates. Other scenarios (Q2 and Q3) forked larger meta-provenance forests and thus spent more time on generating repairs and on solving constraints. However, we observe that, even when run on a single machine, the entire process took less than 25 seconds in all scenarios, which does not seem unreasonable. This time could be further reduced by parallelization, since different machines could work on different parts of the meta-provenance forest in parallel.

5.6 Backtesting speed

Next, we evaluate the backtesting speed using the repair candidates listed in Table 2. For each candidate, we sampled packet traces at the network ingresses from the log, and replayed them for backtesting. The top line in Figure 9b shows the time needed to backtest all the candidates sequentially; testing all nine of them took about two minutes, which already seems reasonably fast. However, the less time backtesting takes, the more repair candidates we can afford to consider. The lower line in Figure 9b shows the time needed to *jointly* backtest the first k candidates using the multi-query optimization technique from Section 4.4, which merges the candidates into a single "backtesting program". With this, testing all nine candidates took about 40 seconds. This large speedup is expected because the repairs are small and fairly similar (since they are all intended to fix the same problem); hence, there is a substantial amount of overlap between the individual backtests, which the multi-query technique can then eliminate.

5.7 Scalability

To evaluate the scalability of meta provenance with regard to the network size, we tested the turnaround time of query Q1 on larger networks which contained up to 169 routers and 549 hosts. We obtained these networks by adding more routers and hosts to the basic Stanford campus network. Moreover, we increased the number of hosts that replay traffic traces [5] to up to 16. We generated synthetic traffic on the remaining hosts, and used higher traffic rates in larger networks to emulate more hosts. As we can see from Figure 9c, the turnaround time increased linearly with the network size, but it was within 50 seconds for all cases. As the breakdown shows, the increase mainly comes from the latency increase of the historical lookups and of the replay. This is because the additional nodes and traffic caused the size of the controller state to increase. This in turn resulted in a longer time to search through the controller state, and to replay the messages. Repair generation and constraint solving time only see minor increases. This is expected because the meta provenance forest is generated from only relevant parts of the log, the size of which is relatively stable when the affected flows are given.

5.8 Applicability to other languages

To see how well meta provenance works for languages other than NDlog, we developed meta models for Trema [51] and Pyretic [37]. This required only a moderate effort (16 person-hours). Our Trema model contains 42 meta rules and 32 meta tuples; it covers basic control flow (e.g., functional calls, conditional jumps) and data flow semantics (e.g., constants, expressions, variables, and objects) of Ruby. The Pyretic model contains 53 meta rules and 41 meta tuples; it describes a set of imperative features of Python, similar to that of Ruby. It also encodes the Pyretic NetCore syntax (from Figure 4 in [37]). Developing such a model is a one-time investment – once rules for a new language are available, they can be applied to any program in that language.

To verify that these models generate effective fixes, we recreated the scenarios in Section 5.3 for Trema and Pyretic. We could not reproduce Q4 in Pyretic because

	Q1	Q2	Q3	Q4	Q5
Trema (Ruby)	7/2	10/2	11/2	10/2	14/3
Pyretic (DSL + Python)	4/2	11/2	9/2	-	14/3

Table 3: Results for Trema and Pyretic. For each scenario from Section 5.3, we show how many repair candidates are generated, and how many passed backtesting.

the Pyretic abstraction and its runtime already prevents such problems from happening. Table 3 shows our results. Overall, the number of repairs that were generated and passed backtesting are relatively stable across the different languages. For Q1, we found fewer repair candidates for Pyretic than for RapidNet and Trema; this is because an implementation of the same logic in different languages can provide different "degrees of freedom" for possible repairs. (For instance, an equality check `Swi==2` in RapidNet would be $match(switch = 2)$ in Pyretic; a fix that changes the operator to $>$ is possible in the former but disallowed in the latter because of the syntax of $match$.) In all cases, meta provenance produced at least one repair that passed the backtesting phase.

6 Related Work

Provenance: Provenance [6] has been applied to a wide range of systems [3, 12, 18, 38, 57]. It has been used for network diagnostics before – e.g., in [10, 55, 62, 63] – but these solutions only explain why some data was or was not computed from some given input data; they do not include the program in the provenance and thus, unlike our approach, cannot generate program fixes. We have previously sketched our approach in [53]; the present paper adds a full algorithm and an experimental evaluation.

Program slicing: Given a specification of the output, program slicing [1, 42, 52] can capture relevant parts of the program by generating a reduced program, which is obtained by eliminating statements from the original program. However, slices do not encode causality and thus cannot be directly used for generating repairs.

Network debugging: There is a rich literature on finding bugs and/or certifying their absence. Some systems, such as [15, 17, 24, 25, 26, 59], use static analysis for this purpose; others, including [46, 47, 56, 58], use dynamic testing. Also, some domain-specific languages can enable verification of specific classes of SDN programs [2, 28, 39]. In contrast, the focus of our work is not verification or finding bugs, but generating *fixes*.

Automated program repair: Tools for repairing programs have been developed in several areas. The software engineering community has used genetic programming [30], symbolic execution [40], and program synthesis [8] to fix programs; they usually rely on a test suite or a formal specification to find fixes and sometimes propose only specific kinds of fixes. In the systems community, ClearView [43] mines invariants in programs, correlates violations with failures, and generates fixes at runtime; ConfDiagnoser [60] compares correct and undesired executions to find suspicious predicates in the program; and Sidiroglou et al. [48] runs attack vectors on instrumented applications and then generates fixes automatically. In databases, ConQueR [50] can refine a SQL query to make certain tuples appear in, or disappear from, the output; however, it is restricted to SPJA queries and cannot handle general controller programs. These systems primarily rely on heuristics, whereas our proposed approach uses provenance to track causality and can thus pinpoint specific root causes.

In the networking domain specifically, the closest solutions are NetGen [45] and Hojjat et at. [22], which synthesize changes to an existing network to satisfy a desired property or to remove incorrect configurations, which are specified as regular expressions or Horn clauses. While these tools can generate optimal changes, e.g., the smallest number of next-hop routing changes, they are designed for repairing the data plane, i.e., a snapshot of the network configuration at a particular time; our approach repairs control programs and considers dynamic network configuration changes triggered by network traffic.

Synthesis: One way to avoid buggy network configurations entirely is to synthesize them from a specification of the operator's intent as, e.g., in Genesis [49]. However, it is unclear whether this approach works well for complex networks or policies, so having a way to find and fix bugs in manually written programs is still useful.

7 Conclusion

Network diagnostics is almost a routine for today's operators. However, most debuggers can only find bugs, but not suggest a fix. In this paper, we have taken a step towards better tool support for network repair, using a novel data structure that we call meta provenance. Like classic provenance, meta provenance tracks causality; but it goes beyond data causality and treats the program as just another kind of data. Thus, it can be used to reason about program changes that prevent undesirable events or create desirable events. While meta provenance falls short of our (slightly idealistic) goal of an automatic "Fix it!" button for SDNs, we believe that it does represent a step in the right direction. As our case studies show, meta provenance can generate high-quality repairs for realistic network problems in one minute, with no help from the human operator.

Acknowledgments: We thank our shepherd Nate Foster and the anonymous reviewers for their comments and suggestions. This work was supported in part by NSF grants CNS-1054229, CNS-1065130, CNS-1453392, CNS-1513679, and CNS-1513734, as well as DARPA/I2O contract HR0011-15-C-0098.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proc. PLDI*, 1990.

[2] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *Proc. POPL*, 2014.

[3] A. Bates, D. Tian, K. R. Butler, and T. Moyer. Trustworthy whole-system provenance for the Linux kernel. In *Proc. USENIX Security*, 2015.

[4] R. Beckett, X. K. Zou, S. Zhang, S. Malik, J. Rexford, and D. Walker. An assertion language for debugging SDN applications. In *Proc. HotSDN*, 2014.

[5] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proc. IMC*, 2010.

[6] P. Buneman, S. Khanna, and T. Wang-Chiew. Why and where: A characterization of data provenance. In *Proc. ICDT*, 2001.

[7] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE way to test openflow applications. In *Proc. NSDI*, 2012.

[8] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *Proc. ICSE*, 2011.

[9] A. Chapman and H. Jagadish. Why not? In *Proc. SIGMOD*, 2009.

[10] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance. In *Proc. SIGCOMM*, 2016.

[11] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, 2008.

[12] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Proc. USENIX Security*, 2011.

[13] R. Durairajan, J. Sommers, and P. Barford. Controller-agnostic SDN debugging. In *Proc. CoNEXT*, 2014.

[14] D. Erickson. The Beacon OpenFlow controller. In *Proc. HotSDN*, 2013.

[15] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proc. NSDI*, 2005.

[16] A. Feldmann, O. Maennel, Z. M. Mao, A. Berger, and B. Maggs. Locating Internet routing instabilities. In *Proc. SIGCOMM*, 2004.

[17] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *Proc. NSDI*, 2015.

[18] A. Gehani and D. Tariq. SPADE: Support for provenance auditing in distributed environments. In *Proc. Middleware*, 2012.

[19] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing one thousand queries with one stone. In *Proc. VLDB*, 2012.

[20] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, M. Musuvathi, Z. Zhang, and L. Zhou. Failure recovery: When the cure is worse than the disease. In *Proc. HotOS*, 2013.

[21] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Proc. NSDI*, 2014.

[22] H. Hojjat, P. Reummer, J. McClurgh, P. Cerny, and N. Foster. Optimizing Horn solvers for network repair. In *Proc. FMCAD*, 2016.

[23] E. Katz-Bassett, H. V. Madhyastha, J. P. John, A. Krishnamurthy, D. Wetherall, and T. Anderson. Studying black holes in the Internet with Hubble. In *Proc. NSDI*, 2008.

[24] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proc. NSDI*, 2013.

[25] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proc. NSDI*, 2012.

[26] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *Proc. NSDI*, 2013.

[27] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proc. ICSE*, 2013.

[28] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable dynamic network control. In *Proc. NSDI*, 2015.

[29] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proc. HotNets*, 2010.

[30] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proc. ICSE*, 2012.

[31] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proc. OSDI*, 2004.

[32] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *Proc. ASID*, 2006.

[33] D. Logothetis, S. De, and K. Yocum. Scalable lineage capture for debugging DISC analysis. Technical Report CSE2012-0990, UCSD, 2012.

[34] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Comm. ACM*, 52(11):87–95, 2009.

[35] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. SIGMOD*, 2002.

[36] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *Proc. SIGCOMM*, 2011.

[37] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *Proc. NSDI*, 2013.

[38] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proc. USENIX ATC*, 2006.

[39] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *Proc. NSDI*, 2014.

[40] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *Proc. ICSE*, 2013.

[41] K. Pan, S. Kim, and E. J. Whitehead Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[42] R. Perera, U. A. Acar, J. Cheney, and P. B. Levy. Functional programs that explain their work. In *Proc. ICFP*, 2012.

[43] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proc. SOSP*, 2009.

[44] RapidNet project web page. `http://netdb.cis.upenn.edu/rapidnet/`.

[45] S. Saha, S. Prabhu, and P. Madhusudan. NetGen: Synthesizing data-plane configurations for network policies. In *Proc. SOSR*, 2015.

[46] C. Scott, A. Panda, V. Brajkovic, G. Necula, A. Krishnamurthy, and S. Shenker. Minimizing faulty executions of distributed systems. In *Proc. NSDI*, 2016.

[47] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker. Troubleshooting blackbox SDN control software with minimal causal sequences. In *Proc. SIGCOMM*, 2014.

[48] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *Proc. IEEE Security and Privacy*, 2005.

[49] K. Subramanian, L. DAntoni, and A. Akella. Genesis: Synthesizing forwarding tables in multi-tenant networks. In *Proc. POPL*, 2017.

[50] Q. T. Tran and C.-Y. Chan. How to conquer why-not questions. In *Proc. SIGMOD*, 2010.

[51] Trema: Full-Stack OpenFlow Framework in Ruby and C, 2019. `https://trema.github.io/trema/`.

[52] M. Weiser. Program slicing. In *Proc. ICSE*, 1981.

[53] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. Automated network repair with meta provenance. In *Proc. HotNets*, 2015.

[54] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. Automated bug removal for software-defined networks. Technical Report MS-CIS-17-02, University of Pennsylvania, 2017.

[55] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo. Diagnosing missing events in distributed systems with negative provenance. In *Proc. SIGCOMM*, 2014.

[56] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling record and replay troubleshooting for networks. In *Proc. USENIX ATC*, 2011.

[57] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. NSDI*, 2012.

[58] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Proc. CoNEXT*, 2012.

[59] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *Proc. NSDI*, 2014.

[60] S. Zhang and M. D. Ernst. Automated diagnosis of software configuration errors. In *Proc. ICSE*, 2013.

[61] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure network provenance. In *Proc. SOSP*, 2011.

[62] W. Zhou, S. Mapara, Y. Ren, Y. Li, A. Haeberlen, Z. Ives, B. T. Loo, and M. Sherr. Distributed time-aware provenance. In *Proc. VLDB*, 2013.

[63] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at Internet-scale. In *Proc. SIGMOD*, 2010.

Delta-net: Real-time Network Verification Using Atoms

Alex Horn
Fujitsu Labs of America

Ali Kheradmand
University of Illinois at Urbana-Champaign

Mukul R. Prasad
Fujitsu Labs of America

Abstract

Real-time network verification promises to automatically detect violations of network-wide reachability invariants on the data plane. To be useful in practice, these violations need to be detected in the order of milliseconds, without raising false alarms. To date, most real-time data plane checkers address this problem by exploiting at least one of the following two observations: (i) only small parts of the network tend to be affected by typical changes to the data plane, and (ii) many different packets tend to share the same forwarding behaviour in the entire network. This paper shows how to effectively exploit a third characteristic of the problem, namely: similarity among forwarding behaviour of packets through *parts* of the network, rather than its entirety. We propose the first provably amortized quasi-linear algorithm to do so. We implement our algorithm in a new real-time data plane checker, Delta-net. Our experiments with SDN-IP, a globally deployed ONOS software-defined networking application, and several hundred million IP prefix rules generated using topologies and BGP updates from real-world deployed networks, show that Delta-net checks a rule insertion or removal in approximately 40 microseconds on average, a more than 10× improvement over the state-of-the-art. We also show that Delta-net eliminates an inherent bottleneck in the state-of-the-art that restricts its use in answering Datalog-style "what if" queries.

1 Introduction

In an evermore interconnected world, network traffic is increasingly diverse and demanding, ranging from communication between small everyday devices to large-scale data centres across the globe. This diversity has driven the design and rapid adoption of new open networking architectures (e.g. [41]), built on programmable network switches, which make it possible to separate the control plane from the data plane. This separation opens up interesting avenues for innovation [37], including rigorous analysis for finding network-related bugs. Finding these bugs *automatically* poses the following challenges.

Since the control plane is typically a Turing-complete program, the problem of automatically proving the presence and absence of bugs in the control plane is generally undecidable. However, the data plane, which is produced by the control plane, can be automatically analyzed. While the problem of checking reachability properties in the data plane is generally NP-hard [34], the problem becomes polynomial-time solvable in the restricted, but not uncommon, case where network switches only forward packets by matching IP prefixes [36]. This theoretical fact helps to explain why *real-time data plane checkers* [27, 25, 55] can often automatically detect violations of network-wide invariants on the data plane in the order of milliseconds, without raising false alarms.

To achieve this, most real-time network verification techniques exploit at least one of the following two observations: (i) only small parts of the network tend to be affected by typical changes to the data plane [27, 25], and (ii) many different packets often share the same forwarding behaviour in the entire network [27, 55]. Both observations are significant because the former gives rise to *incremental network verification* in which only changes between two data plane snapshots are analyzed, whereas the latter means that the analysis can be performed on a representative subset of network packets in the form of *packet equivalence classes* [27, 25, 55].

In spite of these advances, it is so far an open problem how to efficiently handle operations that involve swaths of packet equivalence classes [27]. This is problematic because it limits the real-time analysis of network failures, which are common in industry-scale networks, e.g. [13, 4]. Moreover, it essentially prevents data plane checkers from being used to answer "what if" queries in the style of recent Datalog approaches [17, 33] because these hypothetical scenarios typically involve checking the fate of many or all packets in the entire network.

To address this problem, this paper shows how to effectively exploit a third characteristic of data plane checking, namely: similarity among forwarding behaviour of packets through *parts* of the network, rather than its entirety. We show that our approach addresses fundamental limitations (§ 2) in the design of the currently most advanced data plane checker, Veriflow [27].

In this paper, we propose a new real-time data plane checker, Delta-net (§ 3). Instead of constructing *multiple forwarding graphs* for representing the flow of packets in the network [27], Delta-net incrementally transforms a *single edge-labelled graph* that represents *all* flows of packets in the entire network. We present the first provably amortized quasi-linear algorithm to do so (Theorem 1). Our algorithm incrementally maintains the lattice-theoretical concept of *atoms*: a set of mutually disjoint ranges through which it is possible to analyze all Boolean combinations of IP prefix forwarding rules in the network so that every possible forwarding table over these rules can be concisely expressed and efficiently checked. This approach is inspired by Yang and Lam's atomic predicates verifier [55]. While more general, their algorithm has a quadratic worst-case time complexity, whereas ours is quasi-linear. Since Delta-net's atom representation is based on lattice theory, it can be seen as an abstract domain (e.g. [11]) for analyzing forwarding rules. What makes our abstract domain different from traditional ones is that we dynamically refine its precision so that false alarms never occur.

For our performance evaluation (§ 4), we use data sets comprising several hundred million IP prefix rules generated from the UC Berkeley campus, four Rocketfuel topologies [49] and real-world BGP updates [46]. As part of our experiments, we run SDN-IP [31, 47], one of the most mature and globally deployed software-defined networking applications in the ONOS project [7, 42]. We show that Delta-net checks a rule insertion or removal in tens of microseconds on average, a more than 10× improvement over the state-of-the-art [27]. Furthermore, as an exemplar of "what if" scenarios, we adapt a link failure experiment by Khurshid et al. [27], and show that Delta-net performs several orders of magnitude faster than Veriflow [27]. We discuss related work in § 5.

Contributions. Our main contributions are as follows:

- Delta-net (§ 3), a new real-time data plane checker that incrementally maintains a compact representation about the flows of all packets in the network, thereby supporting a broader class of scenarios and queries.
- new realistic benchmarks (§ 4.2.2) with an open-source, globally deployed SDN application [47].
- experimental results (§ 4.3) that show Delta-net is more than 10× faster than the state-of-the-art in checking rule updates, while also making it now feasible to answer an expensive class of "what if" queries.

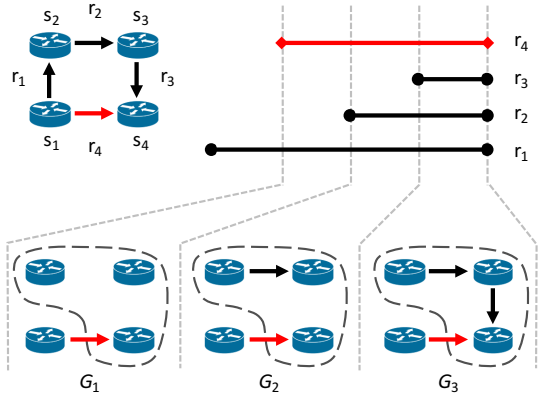


Figure 1: When rule r_4 (red edge) is inserted into switch s_1, Veriflow constructs at least three forwarding graphs, which significantly overlap with each other.

2 Overview of approach

In this section, we motivate and explain our approach through a simple example (§ 2.1) that illustrates how Delta-net differs from the currently most advanced data plane checker, Veriflow [27]. In addition to performance considerations, we follow three design goals (§ 2.2).

2.1 Example

Our example is based on a small network of four switches, shown in the upper-left corner of Figure 1. The data plane in this network is depicted as a directed graph in which each edge denotes an IP prefix forwarding rule. For example, rule r_1 in Figure 1 is assumed to determine the packet flow for a specific destination IP prefix from switch s_1 to s_2. Suppose the network comprises rules r_1, r_2 and r_3 (black edges) installed on switches s_1, s_2 and s_3, respectively. Since each rule matches packets by a destination IP prefix, we can represent each rule's match condition by an interval. For example, the IP prefix 0.0.0.10/31 (using the IPv4 CIDR format) corresponds to the half-closed interval $[10:12) = \{10, 11\}$ because 0.0.0.10/31 is equivalent to the 32-bit binary sequence that starts with all zeros and ends with $101*$ where $*$ denotes an arbitrary bit. Here, we depict the intervals of all three rules as parallel black lines (in an arbitrary order) in the upper-right half of Figure 1. The interpretation is that all three rules' IP prefixes overlap with each other.

Let us assume we are interested in checking the data plane for forwarding loops. Veriflow then first partitions all packets into *packet equivalences classes*, as explained next. Consider a new rule r_4 (red edge in Figure 1) to be installed on switch s_1 such that rule r_4 has a higher priority than the existing rule r_1 on switch s_1. As depicted in the upper half of Figure 1, the new rule r_4 overlaps with

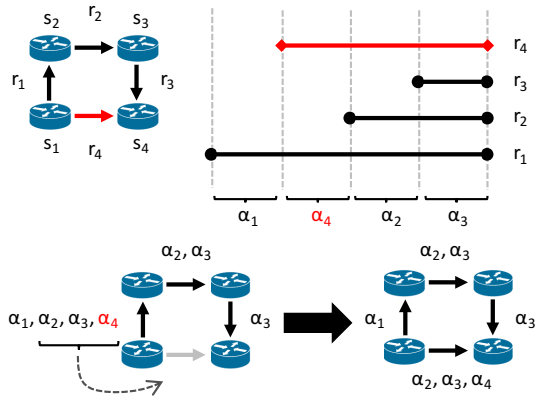


Figure 2: Rather than constructing *multiple* forwarding graphs that potentially overlap (Figure 1), Delta-net incrementally transforms a *single* edge-labelled graph.

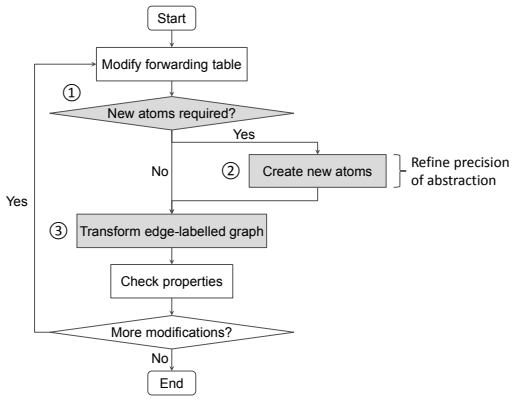


Figure 3: Delta-net incrementally maintains atoms, a family of sets of packets that can represent all Boolean combinations of IP prefix forwarding rules.

all the existing rules in the network, irrespective of the switch on which they are installed. Veriflow identifies at least three equivalence classes that are affected by the new rule, each of which denotes a set of packets that experience the same forwarding behaviour throughout the network. Here, we depict equivalence classes by three interval segments (gray vertical dashed lines).

For each equivalence class, Veriflow constructs a *forwarding graph* (denoted by G_1, G_2 and G_3 in Figure 1) that represent how packets in each equivalence class can flow through the network. Veriflow can now check for, say, forwarding loops by traversing G_1, G_2 and G_3. Note that the edge that represents the packet flow from switch s_1 to s_2 is excluded from all three forwarding graphs because on switch s_1, for the three depicted equivalence classes, the packet flow is determined by the higher-priority rule r_4 rather than the lower-priority rule r_1.

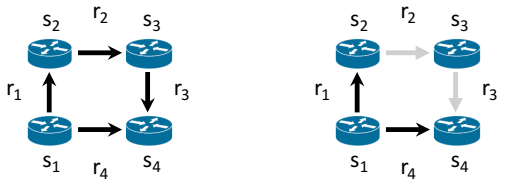
Crucially, in our example, the forwarding graphs that Veriflow constructs are essentially the same to previously constructed ones (dashed areas) except for the new edge from switch s_1 to s_4. In addition, G_1, G_2 and G_2 share much in common, e.g. G_2 and G_3 have the same edge from switch s_2 to s_3. As the number of rules in the network increases, so may the commonality among forwarding graphs. In real networks, this leads to inefficiencies that pose problems under real-time constraints.

We now illustrate how our approach avoids these kind of inefficiencies. For illustrative purposes, assume we start again with the network in which only rules r_1, r_2 and r_3 (black edges) have been installed on switches s_1, s_2 and s_3, respectively. The collection of IP prefixes in the network induces half-closed intervals, each of which we call an *atom*. A set of atoms can represent an IP prefix. For example, as shown at the top of Figure 2, the set $\{\alpha_2, \alpha_3\}$ represents the IP prefix of rule r_2.

At the core of our approach is a directed graph whose edges are labelled by atoms. The purpose of this edge-labelled graph is to represent packet flows in the entire network. For example, to represent that r_2 forwards packets from switch s_2 to s_3 we label the corresponding edge in the directed graph with the atoms α_2 and α_3.

Of course, an edge-labelled graph that represents all flows in the network may need to be transformed when a new rule is inserted or removed. The bottom of Figure 2 illustrates the nature of such a graph transformation in the case where rule r_4 is inserted into switch s_1. The point of the drawing is threefold. First, observe that the rule insertion of r_4 results in the creation of a new atom α_4 (red label in the graph on the bottom-left corner). Using the newly created atom, r_4's IP prefix can now be precisely represented as the set of atoms $\{\alpha_2, \alpha_3, \alpha_4\}$. Second, when a new atom, such as α_4, is created, existing atom representations may need to be updated. For example, r_1's IP prefix on the edge from switch s_1 to s_2 needs to be now represented by four instead of only three atoms. Finally, since rule r_4, recall, has higher priority than rule r_1, three of those four atoms need be moved to the newly inserted edge from switch s_1 to s_4 (as shown by a dashed arrow in Figure 2). This results in the edge-labelled graph shown in the bottom-right corner of Figure 2 where the edges from switch s_1 correspond to the forwarding action of the rules r_1 and r_4 and are labelled by the set of atoms $\{\alpha_1\}$ and $\{\alpha_2, \alpha_3, \alpha_4\}$, respectively. Crucially, note how our approach avoids the construction of *multiple* overlapping forwarding graphs by transforming a *single* edge-labelled graph instead.

Delta-net's key components and sequence of steps are depicted in Figure 3. In this flowchart, the steps in shaded areas — annotated by $\{①, ②\}$ and $\{③\}$ in Figure 3 — are new and described in § 3.1 and § 3.2, respectively. Here, we only highlight two main fundamental differences between Delta-net and Veriflow:



(a) Veriflow (b) Delta-net

Figure 4: Comparison of processed rules (black edges).

- Veriflow generally has to traverse rules in different switches to compute equivalence classes and forwarding graphs: in our example, when rule r_4 is inserted into switch s_1, Veriflow traverses all rules in the network (four black edges in Figure 4a). By contrast, our approach concentrates on the affected rules in the modified switch. For example, when rule r_4 is inserted into switch s_1, the two black edges in Figure 4b show that only rules r_1 and r_4 on switch s_1 are inspected by Delta-net to transform the edge-labelled graph.
- Veriflow recomputes affected equivalence classes and forwarding graphs each time a rule is inserted or removed, whereas Delta-net incrementally transforms a single edge-labelled graph to represent the flows of *all* packets in the entire network. This significantly broadens the scope of Delta-net (§ 2.2) because it can more efficiently handle network failures and "what if" queries regarding *many or all* packets in the network.

2.2 Functional design goals

In addition to more stringent real-time constraints, our work is guided by the following three design goals:

1. Similar to Datalog-based approaches [17, 33], we want to efficiently find *all* packets that can reach a node B from A, avoiding restrictions of SAT/SMT-based data plane checkers (e.g. [34]), which can solve a broader class of problems but require multiple calls to their underlying SAT/SMT solver to find more than one witness for the reachability from A to B.
2. Our design should support known incremental network verification techniques that construct forwarding graphs for the purpose of checking reachability properties each time a rule is inserted or removed [27]. This is important because it preserves one of the main characteristics of previous work, namely: it is practical, and no expertise in formal verification is required to check the data plane.
3. When real-time constraints are less important (as in the case of pre-deployment testing, e.g. [58]), we want to facilitate the answering of a broader class of (possibly incremental) reachability queries, such as *all-pairs reachability* queries in the style of recent Datalog approaches [17, 33]. These kind of queries generally concern the reachability between *all* packets and pairs of nodes in the network. We also aim at efficiently answering queries in scenarios that involve many or all packets, such as link failures [27].

After explaining the technical details of Delta-net, we describe how it achieves these design goals (§ 3.3).

Priority	IP Prefix	Action
High	0.0.0.10/31	drop
Low	0.0.0.0/28	forward

Table 1: A forwarding table for a network switch.

3 Delta-net

In this section, we explain Delta-net's underlying atom representation (§ 3.1), and its algorithm for modifying rules through insertion and removal operations (§ 3.2). Recall that these two subsections correspond to the steps annotated by {①,②} and {③} in Figure 3, respectively.

We illustrate the internal workings of Delta-net using the simple forwarding table in Table 1. It features two rules, r_H and r_L, whose subscript corresponds to their priority: the higher-priority rule, r_H, drops packets whose destination address matches the IP prefix 0.0.0.10/31, whereas the lower-priority rule, r_L, forwards packets destined to the IP prefix 0.0.0.0/28. We elide details about the next hop (where a matched packet should be sent) because it is not pertinent to the example.

As alluded to in the previous section (§ 2.1), we can think of IP prefixes as half-closed intervals: r_H's IP prefix, 0.0.0.10/31, corresponds to the half-closed $[10:12)$. Similarly, $0.0.0.0/28 = [0:16)$ for r_L's IP prefix. Of course, this interval representation can be easily generalized to IPv6 addresses. Next, we show how Delta-net represents rules with such IP prefixes, for some fixed IP address length.

3.1 Atom representation

In this subsection, we describe the concept of atoms; how they are maintained is essential to the rule modifications algorithms in the next subsection (§ 3.2).

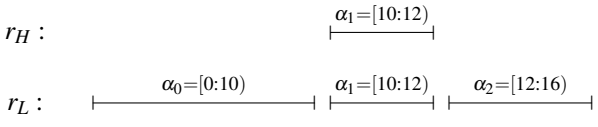


Figure 5: Atoms for the IP prefix rules in Table 1.

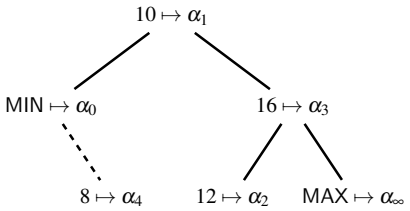


Figure 6: Balanced binary search tree of key/value pairs after inserting the half-closed intervals from Figure 5.

Intuitively, we can segment the IP prefixes of all the rules in the network into disjoint half-closed intervals, which we call atoms. This kind of segmentation is illustrated in Figure 5 using the rules r_H and r_L in Table 1.[1]

By construction of atoms, we can represent an IP prefix of a rule r as a *set of atoms*. We denote this IP prefix representation by $[\![\mathsf{interval}(r)]\!]$. For example, r_H's IP prefix, $[\![\mathsf{interval}(r_H)]\!]$, corresponds to the singleton set consisting of the atom α_1, whereas r_L's IP prefix is $[\![\mathsf{interval}(r_L)]\!] = \{\alpha_0, \alpha_1, \alpha_2\}$. Using these atoms, we can represent, for example, the set difference $[\![\mathsf{interval}(r_L)]\!] - [\![\mathsf{interval}(r_H)]\!]$ to formalize the fact that r_L can only match packets that are not matched by the higher-priority rule r_H. Next, we explain how to devise an efficient representation of atoms such that we can efficiently verify network-wide reachability properties when a rule is inserted or removed (§ 3.2).

At the core of our atom representation is a function, $\mathfrak{M}$, that maps non-negative integers to identifiers. Specifically, $\mathfrak{M}$ is an ordered map that contains key/value pairs $n \mapsto \alpha_i$ where n is a lower or upper bound of an IP prefix of a rule r (denoted by $\mathsf{lower}(r)$ and $\mathsf{upper}(r)$, respectively) and α_i is a unique identifier, called *atom identifier*. For example, $\mathsf{lower}(r_H) = 10$ and $\mathsf{upper}(r_H) = 12$. More generally, we ensure that $\mathsf{MIN} \leq \mathsf{lower}(r) < \mathsf{upper}(r) \leq \mathsf{MAX}$ for every rule r where $\mathsf{MIN} = 0$ and $\mathsf{MAX} = 2^k$ for some fixed positive integer k, e.g. $k = 32$ for 32-bit IP addresses. We maintain the invariant that $\mathfrak{M}$ contains only unique keys. The interpretation of each pair $n \mapsto \alpha_i$ in $\mathfrak{M}$, for all $n < \mathsf{MAX}$, is as follows: the atom identifier α_i denotes the *atom* $[n : n')$ where n' is the next numerically greater key in $\mathfrak{M}$. Each atom identifier, therefore, uniquely denotes a half-closed interval, i.e. an atom. For efficiency reasons, we ensure that each atom identifier is generated from a consecutively increasing counter that starts at zero. Before processing any rules, we initialize $\mathfrak{M}$ by inserting $\mathsf{MIN} \mapsto \alpha_0$ and $\mathsf{MAX} \mapsto \alpha_\infty$ where α_∞ is the greatest atom identifier.

We define the procedure CREATE_ATOMS(r), where $\mathsf{interval}(r) = [\mathsf{lower}(r) : \mathsf{upper}(r))$ is the half-closed interval corresponding to r's IP prefix, such that, if $\mathfrak{M}$ has not already paired $\mathsf{lower}(r)$ with an atom identifier, then it inserts into $\mathfrak{M}$ the key/value pair $\mathsf{lower}(r) \mapsto \alpha_j$ for the next available counter value α_j; similarly, we conditionally insert into $\mathfrak{M}$ the key/value pair $\mathsf{upper}(r) \mapsto \alpha_k$ for the next available counter value α_k. Note that after CREATE_ATOMS(r) has been called, $\mathfrak{M}$ may contain 0, 1, or 2 new atoms (but not more). For example, IP prefixes such as 1.2.0.0/16 and 1.2.0.0/24 have the same lower bound because they only differ in their prefix lengths, and so together yield only three and not four atoms. While the values of atom identifiers depend on the order in which rules are inserted, the set of generated atoms at the end is invariant under the order in which CREATE_ATOMS is called. We also remark that the number of atoms represented by $\mathfrak{M}$ is equal to $\mathfrak{M}$'s size minus one.

For our complexity analysis, we assume that the $\mathfrak{M}$'s insertion and retrieval operations run logarithmically in the size of $\mathfrak{M}$, which could be achieved with a balanced binary-search tree such as a red-black tree. In this case, Figure 6 (excluding the leaf node connected by a dashed edge) illustrates the balanced binary search tree that results after CREATE_ATOMS(r_H) and CREATE_ATOMS(r_L) has been called for the rules r_H and r_L in Table 1. For example, α_1 at the root of the binary search tree in Figure 6 denotes the atom $[10 : 12)$. When clear from the context, we refer to atom identifiers and atoms interchangeably.

3.2 Edge labelling algorithm

Using our atom representation (§ 3.1), we show how to efficiently label the edges of a directed graph that succinctly describes the flow of all packets in the entire network. Our algorithm is incremental in the sense that it only changes edge labels that are affected by the insertion or removal of a rule. Our algorithm, which achieves this incrementality, requires the following notions.

We denote an IP prefix forwarding rule by r, possibly with a prime symbol. Each rule r is associated with $\mathsf{priority}(r)$ and $\mathsf{link}(r)$, as explained in turn. We assume that rules in the same forwarding table whose IP prefixes overlap have pair-wise distinct priorities, denoted by $\mathsf{priority}(r)$.[2] For all rules r and r' in the same forwarding table, r has a *higher priority than* r' if $\mathsf{priority}(r) > \mathsf{priority}(r')$; equivalently, $\mathsf{priority}(r) < \mathsf{priority}(r')$ means that r has a *lower priority than* r'. Note that longest-prefix routing can be simulated by assigning rule priorities according to prefix lengths [55]. We denote by $\mathsf{link}(r)$ a directed edge in a graph that is induced by a network topology. For theoretical and practical reasons (see also § 4.1), $\mathsf{link}(r)$ is purposefully more general than a pair of, say, ports. We write $\mathsf{source}(r)$ for the node

[1] Appendix A illustrates the fact that atoms induce a Boolean lattice.

[2] This assumption is reasonable for, say, OpenFlow tables where the matching of rules with the same highest priority is explicitly undefined.

in the graph on which $\mathsf{link}(r)$ is incident. For example, $\mathsf{source}(r_1) = s_1$ and $\mathsf{source}(r_2) = s_2$ in Figure 2.

From a high-level perspective, Delta-net consists of two algorithms, one for inserting (Algorithm 1) and another for removing (Algorithm 2) a single rule. Both algorithms accesses three global variables: $\mathfrak{M}$, *label* and *owner*, as described in turn. First, $\mathfrak{M}$ is the balanced binary tree described in § 3.1, e.g. Figure 6. Second, given a *link* in the network topology, *label*[*link*] denotes a set of atoms, each of which corresponds to a half-closed interval that a designated field in a packet header h can match for h to be forwarded along the *link*. Finally, *owner* is an array of hash tables, each of which stores a balanced binary search tree containing rules ordered by priority. More accurately, *owner* is an array of sufficient size such that, for every atom α, *owner*$[\alpha]$ is a hash table that maps a *source* node to a balanced binary search tree, *bst*, that orders rules in the *source* node that contain atom α in their interval according to their priority, i.e., we maintain the invariant that *bst* contains only rules r such that *source* $= \mathsf{source}(r)$ and $\alpha \in [\![\mathsf{interval}(r)]\!]$ where *bst* $=$ *owner*$[\alpha]$[*source*]. The highest-priority rule in a non-empty balanced binary search tree *bst* can be retrieved via *bst*.highest_priority_rule(). We remark that we do not use a priority queue because Algorithm 2 described later (§ 3.2.2) needs to be able to remove arbitrary rules, not just the highest-priority one. We write $r \in$ *bst* when rule r is stored in *bst*.

3.2.1 Edge labelling when inserting a rule

We now explain how the INSERT_RULE procedure in Algorithm 1 works. The algorithm starts by calling CREATE_ATOMS$^+$ (line 2) that accomplishes the same as CREATE_ATOMS from § 3.1 except that CREATE_ATOMS$^+$ also returns Δ, a set of *delta-pairs*, as explained next. Each delta-pair in Δ is of the form $\alpha \mapsto \alpha'$ where α and α' are atoms. The intuition is that the half-closed interval previously represented by α needs to be now represented by two atoms instead, namely α and α'. We call this *atom splitting*. In a nutshell, this splitting provides an efficient mechanism for incrementally refining the precision of our abstract domain. This incremental abstraction refinement allows us to precisely and efficiently represent all Boolean combinations of rules in the network (see also § 1).

To illustrate the splitting of atoms, let r_M be a new medium-priority rule to be inserted into Table 1 such that $\mathsf{priority}(r_L) < \mathsf{priority}(r_M) < \mathsf{priority}(r_H)$. Assume r_M's IP prefix is 0.0.0.8/30; hence, $\mathsf{interval}(r_M) = [8 : 12)$. If $\mathfrak{M}$ is the binary search subtree in Figure 6 consisting of undashed edges, then CREATE_ATOMS$^+(r_M)$ returns a single delta-pair, namely $\Delta = \{\alpha_0 \mapsto \alpha_4\}$, where α_0 is the atom identifier denoting the atom $[\mathsf{MIN} : 10)$ before

Algorithm 1 Inserts rule r into a forwarding table.

```
1:  procedure INSERT_RULE(r)
2:    Δ ← CREATE_ATOMS+(r)                          ▷ |Δ| ≤ 2
3:    for α ↦ α′ in Δ do
4:      owner[α′] ← owner[α]
5:      for source ↦ bst in owner[α] do
6:        r′ ← bst.highest_priority_rule()
7:        label[link(r′)] ← label[link(r′)] ∪ {α′}
8:      end for
9:    end for
10:   for α in [[interval(r)]] do
11:     r′ ← null
12:     bst ← owner[α][source(r)]
13:     if not bst.is_empty() then
14:       r′ ← bst.highest_priority_rule()
15:     end if
16:     if r′ = null or priority(r′) < priority(r) then
17:       label[link(r)] ← label[link(r)] ∪ {α}
18:       if r′ ≠ null and link(r) ≠ link(r′) then
19:         label[link(r′)] ← label[link(r′)] − {α}
20:       end if
21:     end if
22:     bst.insert(r)
23:   end for
24: end procedure
```

r_M has been inserted, and α_4 is a new atom identifier, depicted as a dashed leaf in Figure 6. Here, $\Delta = \{\alpha_0 \mapsto \alpha_4\}$ means that the existing atom $[\mathsf{MIN} : 10)$ needs to be split into $\alpha_0 = [\mathsf{MIN} : 8)$ and $\alpha_4 = [8 : 10)$. Note that there are always at most two delta-pairs in Δ. Thus, since $|\Delta| \leq 2$, we can effectively update the atom representation of forwarding rules in an incremental manner.

The splitting of atoms is effectuated by updating the labels for some links in the single-edged graph that represents the flow in the entire network (line 7). To quickly determine these links, we exploit the highest-priority matching mechanism of packets. For this purpose, we use the array of hash tables, *owner*: it associates an atom α and *source* node with a binary search tree *bst* such that *bst*.highest_priority_rule() determines the next hop from *source* of an α-packet (line 6). Since $|\Delta| \leq 2$, the doubly nested loop (line 3–9) runs at most twice. For each delta-pair $\alpha \mapsto \alpha'$ in Δ, the array of hash tables is updated so that *owner*$[\alpha']$ is a copy of *owner*$[\alpha]$ (line 4). Therefore, since $r' \in$ *owner*$[\alpha][\mathsf{source}(r')]$ holds for the existing atom α, it follows that $r' \in$ *owner*$[\alpha'][\mathsf{source}(r')]$ holds for the new atom $\alpha' \in [\![\mathsf{interval}(r')]\!]$, thereby maintaining the invariant of the *owner* array of hash tables (§ 3.2). We adjust the labels accordingly (line 7). The remainder of Algorithm 1 (line 10–23) reassigns atoms based on the priority of the rule that 'owns' each atom, as explained next.

Algorithm 2 Removes rule r from a forwarding table.

```
1:  procedure REMOVE_RULE(r)
2:    for α in [[interval(r)]] do
3:      bst ← owner[α][source(r)]
4:      r' ← bst.highest_priority_rule()
5:      bst.remove(r)
6:      if r' = r then
7:        label[link(r)] ← label[link(r)] − {α}
8:        if not bst.is_empty() then
9:          r'' ← bst.highest_priority_rule()
10:         label[link(r'')] ← label[link(r'')] ∪ {α}
11:       end if
12:     end if
13:   end for
14: end procedure
```

The algorithm continues by iterating over all atoms that collectively represent r's IP prefix (line 10), possibly including the newly created atom(s) in Δ (see previous paragraphs). For each such atom α in $[\![\mathsf{interval}(r)]\!]$, we find the highest-priority rule r' (line 14) that determines the flow of an α-packet at the node $\mathsf{source}(r)$ into which rule r is inserted. We say such a rule r' *owns* α. If no such rule exists or its priority is lower than r's (line 16), we assign α to the set of atoms that determine which network traffic can flow along the link of r (line 17–20), i.e. $label[\mathsf{link}(r)]$. Finally, we insert r into the binary search tree for atom α and node $\mathsf{source}(r)$ (line 22), irrespective of which rule owns atom α.

3.2.2 Edge labelling when removing a rule

Algorithm 2 removes a rule r from a forwarding table. Similar to Algorithm 1, Algorithm 2 iterates over all atoms α that are needed to represent r's IP prefix (line 2). For each such atom α, it retrieves the *bst* that is specific to the node from which r should be removed (line 3). After finding the highest-priority rule r' in *bst* (line 4), it removes r from *bst* (line 5). If r' equals r (line 6), we need to remove α from the label of $\mathsf{link}(r)$ because the rule that needs to be removed, r, owns atom α (as described in § 3.2.1). In addition, we may need to transfer the ownership of the next higher priority rule (line 8-11).

We remark that after the removal of a rule, it may be that some (at most two) atoms are not needed any longer. In this case, akin to garbage collection, we could reclaim the unused atom identifier(s). This 'garbage collection' mechanism is omitted from Algorithm 2.

3.2.3 Complexity analysis

We now show that each rule update is amortized linear time in the number of affected atoms and logarithmic in the maximum number of overlapping rules in a single switch. While in the worst-case there are as many atoms as there are rules in the network, our experiments (§ 4) show that the number of atoms is typically much smaller in practice, explaining why we found Delta-net to be highly efficient in the vast majority of cases.

Theorem 1 (Asymptotic worst-case time complexity)**.** *To insert or remove a total of R rules, Algorithm 1 and 2 have a $O(RK \log M)$ worst-case time complexity where K is the number of atoms and M is the maximum number of overlapping rules per network switch.*

Proof. The proof can be found in Appendix B. □

The space complexity of Delta-net is $O(RK)$ where R and K are the total number of rules and atoms, respectively. We recall that K is significantly smaller than R. We also experimentally quantify memory usage (§ 4).

3.3 Revisited: functional design goals

From a functionality perspective, recall that our work is guided by three design goals (§ 2.2). In this subsection, we explain how Delta-net achieves these goals.

API for persistent network-wide flow information. Delta-net provides an exact representation of all flows through the entire network. For this purpose, Delta-net maintains the atom labels for every edge in the graph that represents the network topology. From a programmer's perspective, this edge-centric information can be always retrieved in constant-time through $label[link]$ where *link* is a pair of nodes in this graph. This way, our API allows a programmer to answer reachability questions about packet flow through the entire network irrespective of the rule that has been most recently inserted or removed. This makes Delta-net different from Veriflow [27]. Architecturally, our generalization is achieved by decoupling packet equivalence classes (whether affected by a rule update or not) from the construction of their corresponding forwarding graphs, cf. [27].

Incremental network verification via delta-graphs. Similar to Veriflow [27], Delta-net can build forwarding graphs, if necessary, to check reachability properties that are suitable for incremental network verification, such as checking the existence of forwarding loops each time a rule is inserted or removed. In fact, the concept of atoms has as consequence a convenient algorithm for computing a compact edge-labelled graph, called *delta-graph*, that represents all such forwarding graphs. We can generate a delta-graph as a by-product of Algorithm 1 for all atoms α whose owner changes (line 16-21); similarly for Algorithm 2. If so desired, multiple rule updates may be aggregated into a delta-graph.

Algorithm 3 Compute all-pairs reachability of all atoms.

```
1: for k, i, j in V do                          ▷ Triple nested loop
2:     label[i, j] ← label[i, j] ∪ (label[i,k] ∩ label[k, j])
3: end for
```

Easier checking of other reachability properties. Delta-net's design provides a lattice-theoretic foundation for transferring known algorithmic techniques to the field of network verification. For example, Algorithm 3 adapts the Floyd–Warshall algorithm to compute the transitive closure of packet flows between all pairs of nodes in the network. Note that our adaptation interchanges the usual maximum and addition operators with union and intersection of sets of atoms, respectively. This way, Algorithm 3 process multiple packet equivalence classes in each hop.[3] Veriflow has not been designed for such computations, and Algorithm 3 illustrates how Delta-net facilitates use cases beyond the usual reachability checks, cf. [27, 25, 55]. This algorithm could be run either on the edge-labelled graph that represents the entire network or only its incremental version in form of a delta-graph (see previous paragraph).

While decision problems such as all-pairs reachability have a higher computational complexity (e.g., Algorithm 3's complexity is $O(K|V|^3)$ where K and V is the number of atoms and nodes in the edge-labelled graph, respectively), they are relevant and useful during pre-deployment testing of SDN applications, as demonstrated by recent work on Datalog-based network verification, e.g. [17, 33]. The fact that our design makes it possible to verify network-wide reachability by intersecting or taking the union of sets of atoms [55] is also relevant for scenarios that involve many or all packet equivalence classes at a time, such as "what if" queries, network failures, and traffic isolation properties, e.g. [3, 18].

4 Performance evaluation

In this section, we experimentally evaluate our implementation of Delta-net (§ 4.1) on a diverse range of data sets (§ 4.2) that are significantly larger than previous ones (see also Appendix C). Our experiments provide strong evidence that Delta-net significantly advances the field of real-time network verification (§ 4.3).

4.1 Implementation

We implemented Algorithm 1 and 2 in C++14 [22]. Our implementation is single-threaded and comprises around 4,000 lines of code that only depend on the C++14 standard library. In particular, we use the standard hashmap, balanced binary search tree and resizeable array implementations. We implement edge labels as customized dynamic bitsets, stored as aligned, dynamically allocated, contiguous memory. We detect forwarding loops via an iterative depth-first graph traversal.

We remark that while Algorithm 1 and 2 focus on handling IP prefix rules, our approach can be extended for other packet header fields. For non-wildcard (i.e. concrete) header fields, our implementation achieves this by encoding composite match conditions as separate nodes in the single edge-labelled graph. For example, if a switch s contains rules that can match three input ports, we encode s as three separate nodes in the edge-labelled graph. It is for this reason that we report the number of graph nodes rather than the number of switches when describing our data sets in the next subsection.

[3] A routine proof by induction on k (the outermost loop) shows that Algorithm 3 computes the all-pairs reachability of every α-packet.

Data set	Nodes	Max Links	Operations
Berkeley	23	252	25.6×10^6
INET	316	40,770	249.5×10^6
RF 1755	87	2,308	67.5×10^6
RF 3257	161	9,432	149.0×10^6
RF 6461	138	8,140	150.0×10^6
Airtel 1	68	260	14.2×10^6
Airtel 2	68	260	505.2×10^6
4Switch	12	16	1.12×10^6

Table 2: Data sets used for evaluating Delta-net.

4.2 Description of data sets

Our data sets are publicly available [14] and can be broadly divided into two classes: data sets derived from the literature (§ 4.2.1), and data sets gathered from an ONOS SDN application (§ 4.2.2). Both are significant as the former avoids experimental bias, whereas the latter increases the realism of our experiments. To achieve reproducibility, we organize our data sets as text files in which each line denotes an *operation*: an insertion or removal of a rule. So all operations can be easily replayed.

Table 2 summarizes our data sets in terms of three metrics.The second and third column in Table 2 correspond to the maximum number of nodes and links in the edge-labelled graph, respectively. We recall that the number of nodes is proportional to the number of ports and switches in the network (§ 4.1). The total number of operations is reported in the last column. Note that most of our data sets are significantly larger than previous ones, cf. [27, 10, 25, 55] (see also Appendix C). Next, we describe the main features of our data sets.

4.2.1 Synthetic data sets

To avoid experimental bias, our experiments purposefully include data sets from the literature [59, 39] that

feature network topologies from the UC Berkeley campus and the Rocketfuel (RF) project [49], namely ASes 1755, 1239, 6257 and 6461. Note that the RF topologies in [39] correspond to those used by [21, 19, 51]. For each of these five network topologies, we generate forwarding rules following the same mechanism as in [59], namely: we gather IP prefixes from over a half a million of real-world BGP updates collected by the Route Views project [46], and compute the shortest paths in a network topology [30]. For example, for the network topology RF 1239, this results in the INET data set [59], a synthetic wide-area backbone network that contains approximately 300 routers, 481 thousand subnets and 125 million IPv4 forwarding rules. We modify the data sets so that rules are inserted with a random priority. After rules have been inserted, we remove them in random order. The first five rows in Table 2 show the resulting data sets, which contain up to 125 million rules. Due to rule removals, the total number of operations is twice the maximum number of rules. Collectively, the Berkeley, INET and RF 1755, 3257 and 6461 data sets comprise around 640 million rule operations. Next, we explain the remaining three data sets in Table 2.

4.2.2 SDN-IP Application

In addition to synthetic data sets (§ 4.2.1), we run experiments with ONOS [7, 42], an open SDN platform used by sizeable operator networks around the globe [7, 42].

To obtain a relevant and realistic experimental setup, we run SDN-IP [31, 47], an important ONOS application that allows an ONOS-controlled network to interoperate with external autonomous networks (ASes). This interoperability is achieved as follows (Figure 7). Inside the ONOS-controlled network reside *Border Gateway Protocol* (BGP) speakers (in our experimental setup there is exactly one internal BGP speaker) that use eBGP to exchange BGP routing information with the border routers of adjacent external ASes. This information, in turn, is propagated inside the ONOS-controlled network via iBGP. As sketched in the upper half of Figure 7, SDN-IP listens to these iBGP messages and requests ONOS to dynamically install IP forwarding rules such that packets destined to an external AS arrive at the correct border router. In doing so, SDN-IP sets the priority of rules according to the longest prefix match where rules with longer prefix lengths receive higher priority. For each rule insertion and removal (depicted by $+r_1$ and $-r_2$ in Figure 7), Delta-net checks the resulting data plane.

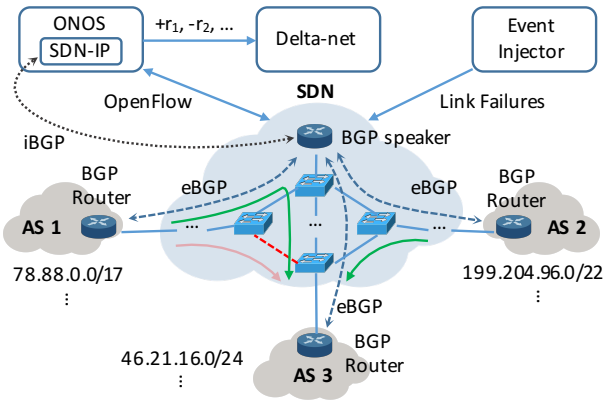


Figure 7: Experimental setup with SDN-IP application.

For our experiments, we run SDN-IP in a single ONOS instance. We use Mininet [29] to emulate a network of sixteen Open vSwitches [43], configured according to the Airtel network topology (AS 9498) [28]. We connect each of these OpenFlow-compliant switches [38] to an external border router that we emulate using Quagga [45]. We configure Quagga such that each border router advertises one hundred IP prefixes, which we randomly select from over half a million real-world IP prefixes gathered from the Route Views project [46], resulting in a total of 1,600 unique (but possibly overlapping) IP prefixes.

Our experiments in § 4.3.1 exploit the fact that SDN-IP relies on ONOS to reconfigure the OpenFlow switches when parts of the network fail. Since network failures happen frequently [4] and pose significant challenges for real-time data plane checkers [25, 27], we can generate interesting data sets by systemically failing links, controlled by the 'Event Injector' process in the upper right half of Figure 7. In particular, the Airtel 1 data set contains the rule insertions and removals triggered by failing a single inter-switch link at a time, recovering each link before failing the next one. Such a link failure (dashed red edge) is illustrated in the left half of Figure 7, causing ONOS to reconfigure the data plane so that a new path is established (green arrow on the left) that avoids the failed link, which caused disruption to earlier network traffic (red arrow). In the case of Airtel 2, we automatically induce all 2-pair link failures (separately failing the first link and then the second one), including their recovery.

We also wanted to study a larger number of rules and IP prefixes, but were limited due to technical issues with ONOS. We worked around these limitations by using a 4-switch ring network. In this smaller ring topology, we configure each Quagga instance to advertise 5,000 IP prefixes (rather than only 100 IP prefixes as in the Airtel experiments), again randomly selected from the Route Views project [46]. We do not fail any links. Instead, we only collect the rules generated by SDN-IP, a process we repeat fourteen times with different IP prefixes. This workaround yields the 4Switch data set in Table 2, comprising 1.12 million rules. In contrast to the previously

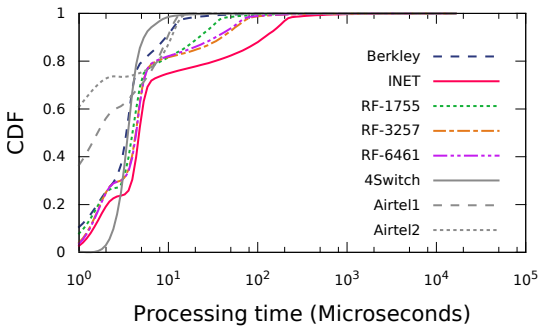


Figure 8: CDF of combined time (microseconds) for processing a rule update and checking for forwarding loops.

described data sets, all of the operations in the 4Switch data set are rule insertions.

4.3 Experimental results

Our experiments separately measure Delta-net's performance in checking individual rule updates (§ 4.3.1) and handling a "what if" scenario (§ 4.3.2). In both cases, at the cost of higher memory usage, Delta-net is more than 10× faster than the state-of-the-art. We run our experiments on an Intel Xeon CPU with 3.47 GHz and 94 GB of RAM. Since our implementation is single-threaded (§ 4.1), we utilize only one out of the 12 available cores.

4.3.1 Checking network updates

To evaluate Delta-net's performance with respect to rule insertions and removals, we build the delta-graph (§ 3.3) for each operation, and find in it all forwarding loops, a common network-wide invariant [26, 25, 55, 27, 59]. We process the rules in each data set in the order in which they appear in the data sets (§ 4.2).

Table 3 summarizes our experimental results for measuring the checking of rule insertions and removals. The first row in Table 3 shows that the total number of atoms is much smaller than the total number of rules in the network (recall Table 2), suggesting a significant degree of commonality among IP prefix rules that atoms effectively exploit. Furthermore, for all data sets, the median and average rule processing time is less than 5 and 41 microseconds, respectively, which includes the checking of forwarding loops. On closer inspection, as shown in the last row of Table 2, Delta-net processes rule updates and checks for the existence of forwarding loops in less than 250 microseconds for at least 98.5% of cases. The combined time for processing a rule update and finding *all* forwarding loops in the corresponding delta-graph (§ 3.3) is visualized by the cumulative density function (CDF) in Figure 8. It shows that the INET data set [59] (solid red line) is one of the more difficult ones for Delta-net. We remark that Delta-net's memory usage never exceeds the available memory on our machine (Appendix D).

Our measurements are significant because earlier experiments with Veriflow [27] result in an average verification time of 380 microseconds, whereas Delta-net verifies rule insertions and removals in often tens of microseconds, and 41 microseconds on average even on the largest data set, INET. This comparison is meaningful because our data sets are significantly larger than previous ones [27, 10, 25, 55]. Moreover, two of our data sets (Airtel 1 and 2) are derived from a real-world software-defined networking application while causing an extensive number of link failures in the network, which were previously shown to lead to longer verification times [25, 27]. Our experiments therefore provide strong evidence that Delta-net can be at least one order of magnitude faster compared to Veriflow [27]. Since neither Veriflow's implementation (or its algorithm) nor any of the data sets used for its experimental evaluation are publicly available, and neither its time nor space complexity is specified, we further quantify the differences between Delta-net and Veriflow by re-implementing a consistent interpretation of Veriflow, as described next.

Our re-implementation of Veriflow, which we call *Veriflow-RI*, is not intended to be a full-feature copy of Veriflow, but rather a re-implementation of their core idea to enable an honest comparison with Delta-net. Specifically, Veriflow-RI is designed for matches against a single packet header field. This explains why Veriflow-RI uses a one-dimensional trie data structure in which every node has at most two children (rather than three [27]). We optimize the computation of equivalence classes and construction of forwarding graphs. Note that these optimizations may not be possible in the original Veriflow implementation with its ternary trie data structure, and Veriflow-RI may therefore be faster than Veriflow [27]. We remark that Veriflow-RI's space complexity is linear in the number of rules in the network, whereas its time complexity is quadratic, rather than quasi-linear as in the case of Delta-net (Theorem 1).

While Delta-net is only approximately 4× faster than Veriflow-RI on the Airtel data set, on the INET data set, Delta-net is approximately 6× faster than Veriflow-RI. This gap widens on the RF 3257 and 6461 data sets where Delta-net is approximately 7× faster than Veriflow-RI. In turn, however, Veriflow-RI consumes $5-7\times$ less memory than Delta-net (Appendix § D).

It is therefore natural to ask whether this trade-off in space and time is worth it. Next, we answer this question affirmatively by showing that Delta-net can check properties for which Veriflow often times out. This difference in run-time performance is due to the fact that Delta-net incrementally maintains flow information of every packet in the entire network, whereas Veriflow recom-

	Berkeley	INET	RF 1755	RF 3257	RF 6461	Airtel 1	Airtel 2	4Switch
Total number of atoms	668,520	563,480	726,535	726,535	726,535	2,799	2,799	443,443
Median rule processing time	4 μs	5 μs	4 μs	5 μs	5 μs	2 μs	1 μs	4 μs
Average rule processing time	5 μs	41 μs	11 μs	22 μs	20 μs	3 μs	3 μs	5 μs
Percentage $< 250\,\mu s$	99.9%	98.5%	99.8%	99.6%	99.7%	99.9%	99.9%	99.9%

Table 3: Experimental results using Delta-net, measuring rule insertions and removals.

putes the forwarding graph for each affected equivalence class. What is remarkable is that Delta-net achieves this extra bookkeeping without limiting the checking of individual network updates (see previous paragraph).

4.3.2 Beyond network updates

We show how Delta-net can go beyond traditional data plane checks per network update. To do so, we consider the following question, which was previously posed by [27], as an exemplar of a "what if" query: What is the fate of packets that are using a link that fails? We interpret their question to mean that Veriflow has to construct forwarding graphs for all packet equivalence classes that are affected by a link failure. This is known to be a difficult task for Veriflow since it requires the construction of at least a hundredfold more forwarding graphs compared to checking a rule insertion or removal (§ 4.3.1). Here, our experiment quantifies how much Delta-net gains by incrementally transforming a single-edge labelled graph instead of constructing multiple forwarding graphs.

For our experiments, we generate a consistent data plane from all the rule insertions in the five synthetic and 4Switch data sets in Table 2, respectively. And in the case of Airtel, we extract a consistent data plane snapshot from ONOS. The total number of resulting rules in each data plane is shown in the second column of Table 4. For all of these seven data planes, we answer which packets and parts of the network are affected by a hypothetical link failure. The verification task therefore is to represent via one or multiple graphs all flows of packets through the network that would be affected when a link fails. The third column in Table 2 (number of links) corresponds to the number of queries we pose, except for the new Airtel data plane snapshot where we pose 158 queries.

Since Delta-net already maintains network-wide packet flow information, we expect it to perform better than Veriflow-RI.[4] The third and fourth column in Table 4 quantify this performance gain by showing the average query time of Veriflow-RI and Delta-net, respectively. On three data planes, Veriflow-RI exceeds the total run-time limit of 24 hours, whereas the longest running Delta-net experiment takes a total of 3.2 hours. When these time outs in Veriflow-RI occur, we report its incomplete average query time t as '$t^{\dagger}$'. We find that Delta-net is usually more than 10× faster than Veriflow-RI (even if Delta-net checks for forwarding loops, as reported in the last column). Since Delta-net is very fast in maintaining the flow of packets, the difference between the last two columns in Table 4 shows that Delta-net's processing time is dominated by the property check (here, forwarding loops). In contrast to Delta-net, Veriflow's processing time is reportedly dominated by the construction of forwarding graphs [27].

[4] Recall from previous experiments (§ 4.3.1), Delta-net's extra bookkeeping poses no performance problems for checking network updates.

5 Related work

In this section, we discuss related works in the literature.

Stateful networks. One of the earliest stateful network analysis techniques [9] proposes symbolic execution of OpenFlow applications using a simplified model of OpenFlow network switches. VeriCon [5] uses an SMT solver to automatically prove the correctness of simple SDN controllers. FlowTest [15] investigates relevant AI planning techniques. SymNet [50] symbolically analyzes stateful middleboxes through additional fields in the packet header. Unlike [9], BUZZ [16] adopts a symbolic model-based testing strategy [52] as a way to capture the state of forwarding devices. Most recent complexity results [53] are the first step towards a taxonomy of decision procedures in this research area. Real-time network verification techniques (see next paragraph) can be extended to check safety properties that depend on the state of the SDN controller [6].

Stateless networks. The seminal work of Xie et al. [54] introduces stateless data plane checking to which Delta-net belongs. The research that emerged from [54] can be broadly divided into offline [57, 2, 24, 40, 1, 34,

Data plane	Rules	Average query time (*ms*)		
		Veriflow-RI	Delta-net	+Loops
Berkeley	12,817,902	3,073.0	**4.7**	93.3
INET	124,733,556	29,117.5†	**0.7**	2,888.6
RF 1755	33,732,869	8,100.6	**1.3**	897.4
RF 3257	74,492,920	17,645.3†	**1.0**	2.6
RF 6461	75,005,738	17,594.5†	**0.4**	0.4
Airtel	38,100	4.5	**0.04**	2.3
4Switch	1,120,000	433.4	**21.1**	128.1

Table 4: Experimental results for "what if" link failures.

48, 26, 35, 17, 33] and online [27, 25, 55] approaches. The offline approaches encode the problem into Datalog [17, 33] or logic formulas that can be checked for satisfiability by constructing a Binary Decision Diagram [57, 2] or calling an SAT/SMT solver [24, 40, 1, 34, 48, 23, 35]. By contrast, all modern online approaches [27, 25, 55] partition in some way the set of all network packets. In particular, the partitioning scheme described in [26], on which [27] is based, dynamically computes equivalence classes by propagating ternary strings in the network, whereas more recent work [25, 55, 8], including ours, pre-compute network packet partitions prior to checking a verification condition. Our work could be used in conjunction with network symmetry reduction techniques [44]. Custom network abstractions can be very useful for restricted cases [20]. While potentially less efficient, our work is more general than [20], and most closely related to [27, 10, 25, 55, 59, 8], which we discuss in turn. The complexity of the most prominent of these works, including Veriflow [27] and NetPlumber [25], is summarized in work [32, Section II] that is independent from ours.

Veriflow [27] constructs multiple forwarding graphs that may significantly overlap (§ 2.1). Our algorithm exploits this overlapping and transforms a single edge-labelled graph instead. Moreover, Veriflow relies on the fact that overlapping IP prefixes can be efficiently found using a trie data structure [27]. By contrast, atoms are generally not expressible as a single IP prefix. For example, atom $[0:10)$ in Figure 5 can only be represented by the union of at least two IP prefixes.

Chen [10] shows how to optimize Veriflow [27], while retaining its core algorithm. Similar to [10], we represent IP prefixes in a balanced binary search tree. Unlike [10], however, our representation serves as a built-in index of half-closed intervals through which we address fundamental limitations of Veriflow (§ 2.1).

NetPlumber [25] incrementally creates a graph that, in the worst case, consists of R^2 edges where R is the number of rules in the network. In contrast to NetPlumber, Delta-net maintains a graph whose size is proportional to the number of links in the network, which is usually much smaller than R. Since the number of atoms tends to be much less than R (§ 4), Delta-net has an asymptotically smaller memory footprint than NetPlumber.

Yang and Lam [55] propose a more compact representation of forwarding graphs that reduces the task of data plane checking to intersecting sets of integers. For the restricted, but common, case of checking IP forwarding rules, our algorithm is asymptotically faster than theirs. Our algorithm, however, does not find the unique minimal number of packet equivalence classes, cf. [55].

More recent work for stateless and non-mutating data plane verification [8] encodes a canonical form of ternary bit-vectors, and shows on small data sets with a few thousand rules that their encoding performs better than Yang and Lam [55]'s algorithm. It would be interesting to repeat these experiments on our, significantly larger, data sets.

Finally, Libra [59] may be used for incrementally checking network updates, but it requires an in-memory "streaming" MapReduce run-time, whereas Delta-net avoids the overheads of such a distributed system. Since Libra's partitioning scheme into disjoint subnets is orthogonal to our algorithm, however, it would be interesting to leverage both ideas together in future work.

6 Concluding remarks

In this paper, we presented Delta-net (§ 3), a new data plane checker that is inspired by program analysis techniques in the sense that it automatically refines a lattice-theoretical abstract domain to precisely represent the flows of all packets in the entire network. We showed that this matters from a theoretical and practical point of view: Delta-net is asymptotically faster and/or more space efficient than prior work [27, 25, 55], and its new design facilitates Datalog-style use cases [17, 33] for which the transitive closure of many or all packet flows needs to be efficiently computed (§ 3.3). In addition, Delta-net can be used to analyze catastrophic network events, such as link failures, for which current incremental techniques are less effective. To show this experimentally (§ 4), we ran an adaptation of the link failure experiments by Khurshid et al. [27] on data sets that are significantly larger than previous ones. For this exemplar "what if" scenario, we found that Delta-net is several orders of magnitude faster than the state-of-the-art (Table 4). Our work therefore opens up interesting new research directions, including testing scenarios under different combinations of failures, which have been shown to be effective for distributed systems, e.g. [56].

Future work. One advantage of Delta-net is that its main loops over atoms in Algorithm 1 and 2 are highly parallelizable. In addition, (stateless) packet modification of IP prefixes can be easily supported without substantial changes to the data structures by augmenting the edge-labelled graph with the necessary information on how atoms are transformed along hops. We are also studying an improved version of Delta-net that avoids the quadratic space complexity by exploiting properties of IP prefixes. Finally, since a naive implementation of Delta-net is exponential in the number of range-based packet header fields (as is Veriflow's [32, Section II]), it would be interesting to guide further developments into multi-range support in higher dimensions using the 'overlapping degree' among rules [32].

Acknowledgements. We would like to thank Sho Shimizu, Pingping Lin and members of the ONOS developer mailing list for technical support. We thank Rao Palacharla, Nate Foster and Mina Tahmasbi for their invaluable feedback on an early draft of this paper. We also would like to thank Ratul Mahajan and the anonymous reviewers of NSDI for their detailed comments and helpful suggestions.

References

[1] AL-SHAER, E., AND AL-HAJ, S. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *SafeConfig* (2010).

[2] AL-SHAER, E., MARRERO, W., EL-ATAWY, A., AND EL-BADAWI, K. Network configuration in a box: towards end-to-end verification of network reachability and security. In *ICNP* (2009).

[3] ANDERSON, C. J., FOSTER, N., GUHA, A., JEANNIN, J.-B., KOZEN, D., SCHLESINGER, C., AND WALKER, D. NetKAT: Semantic foundations for networks. In *POPL* (2014).

[4] BAILIS, P., AND KINGSBURY, K. The network is reliable. *Queue 12*, 7 (July 2014), 20:20–20:32.

[5] BALL, T., BJØRNER, N., GEMBER, A., ITZHAKY, S., KARBYSHEV, A., SAGIV, M., SCHAPIRA, M., AND VALADARSKY, A. VeriCon: Towards verifying controller programs in software-defined networks. In *PLDI* (2014).

[6] BECKETT, R., ZOU, X. K., ZHANG, S., MALIK, S., REXFORD, J., AND WALKER, D. An assertion language for debugging SDN applications. In *HotSDN* (2014).

[7] BERDE, P., GEROLA, M., HART, J., HIGUCHI, Y., KOBAYASHI, M., KOIDE, T., LANTZ, B., O'CONNOR, B., RADOSLAVOV, P., SNOW, W., AND PARULKAR, G. ONOS: Towards an open, distributed SDN OS. In *HotSDN* (2014).

[8] BJØRNER, N., JUNIWAL, G., MAHAJAN, R., SESHIA, S. A., AND VARGHESE, G. ddNF: An efficient data structure for header spaces. In *HVC* (2016).

[9] CANINI, M., VENZANO, D., PEREŠÍNI, P., KOSTIĆ, D., AND REXFORD, J. A NICE way to test openflow applications. In *NSDI* (2012).

[10] CHEN, Z. Veriflow system analysis and optimization. Master's thesis, University of Illinois Urbana-Champaign, 2014.

[11] COUSOT, P., AND COUSOT, R. Systematic design of program analysis frameworks. In *POPL* (1979).

[12] DAVEY, B. A., AND PRIESTLEY, H. A. *Introduction to Lattices and Order*, second ed. Cambridge University Press, 2002.

[13] DEAN, J. Underneath the covers at Google, 2008. Google I/0.

[14] DELTA-NET. https://github.com/delta-net/datasets.

[15] FAYAZ, S. K., AND SEKAR, V. Testing stateful and dynamic data planes with FlowTest. In *HotSDN* (2014).

[16] FAYAZ, S. K., YU, T., TOBIOKA, Y., CHAKI, S., AND SEKAR, V. BUZZ: Testing context-dependent policies in stateful networks. In *NSDI* (2016).

[17] FOGEL, A., FUNG, S., PEDROSA, L., WALRAED-SULLIVAN, M., GOVINDAN, R., MAHAJAN, R., AND MILLSTEIN, T. A general approach to network configuration analysis. In *NSDI* (2015).

[18] FOSTER, N., KOZEN, D., MILANO, M., SILVA, A., AND THOMPSON, L. A coalgebraic decision procedure for NetKAT. In *POPL* (2015).

[19] FRENETIC TOPOLOGIES. https://github.com/frenetic-lang/pyretic/tree/master/pyretic/evaluations. Tree ac942315136e.

[20] GEMBER-JACOBSON, A., VISWANATHAN, R., AKELLA, A., AND MAHAJAN, R. Fast control plane analysis using an abstract representation. In *SIGCOMM* (2016).

[21] HARTERT, R., VISSICCHIO, S., SCHAUS, P., BONAVENTURE, O., FILSFILS, C., TELKAMP, T., AND FRANCOIS, P. A declarative and expressive approach to control forwarding paths in carrier-grade networks. In *SIGCOMM* (2015).

[22] ISO. *International Standard ISO/IEC 14882:2014(E) Programming Language C++*. 2014.

[23] JAYARAMAN, K., BJØRNER, N., OUTHRED, G., AND KAUFMAN, C. Automated analysis and debugging of network connectivity policies. Tech. rep., Microsoft Research, 2014.

[24] JEFFREY, A., AND SAMAK, T. Model checking firewall policy configurations. In *POLICY* (2009).

[25] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *NSDI* (2013).

[26] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: Static checking for networks. In *NSDI* (2012).

[27] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying network-wide invariants in real time. In *NSDI* (2013).

[28] KNIGHT, S., NGUYEN, H., FALKNER, N., BOWDEN, R., AND ROUGHAN, M. The internet topology zoo. *IEEE Journal on Selected Areas in Communications 29*, 9 (Oct. 2011), 1765–1775.

[29] LANTZ, B., HELLER, B., AND MCKEOWN, N. A network in a laptop: Rapid prototyping for software-defined networks. In *SIGCOMM Workshop on Hot Topics in Networks* (2010).

[30] LIBRA. https://github.com/jvimal/libra-data.

[31] LIN, P., HART, J., KRISHNASWAMY, U., MURAKAMI, T., KOBAYASHI, M., AL-SHABIBI, A., WANG, K.-C., AND BI, J. Seamless interworking of SDN and IP. In *SIGCOMM* (2013).

[32] LINGUAGLOSSA, L. *Two challenges of Software Networking: Name-based Forwarding and Table Verification*. PhD thesis, Paris Diderot University, France, 2016.

[33] LOPES, N. P., BJØRNER, N., GODEFROID, P., JAYARAMAN, K., AND VARGHESE, G. Checking beliefs in dynamic networks. In *NSDI* (2015).

[34] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P. B., AND KING, S. T. Debugging the data plane with Anteater. In *SIGCOMM* (2011).

[35] MALDONADO-LOPEZ, F. A., CALLE, E., AND DONOSO, Y. Detection and prevention of firewall-rule conflicts on software-defined networking. In *RNDM* (2015).

[36] MCGEER, R. Verification of switching network properties using satisfiability. In *ICC* (2012).

[37] MCKEOWN, N. How SDN will shape networking, 2011. Open Networking Summit.

[38] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review 38*, 2 (Mar. 2008), 69–74.

[39] NARAYANA, S., TAHMASBI, M., REXFORD, J., AND WALKER, D. Compiling path queries. In *NSDI* (2016).

[40] NELSON, T., BARRATT, C., DOUGHERTY, D. J., FISLER, K., AND KRISHNAMURTHI, S. The Margrave tool for firewall analysis. In *LISA* (2010).

[41] NUNES, B. A. A., MENDONCA, M., NGUYEN, X. N., OBRACZKA, K., AND TURLETTI, T. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys Tutorials 16*, 3 (2014), 1617–1634.

[42] ONOS DEPLOYMENTS. https://wiki.onosproject.org/display/ONOS/Global+SDN+Deployment+Powered+by+ONOS.

[43] PFAFF, B., PETTIT, J., KOPONEN, T., JACKSON, E. J., ZHOU, A., RAJAHALME, J., GROSS, J., WANG, A., STRINGER, J., SHELAR, P., AMIDON, K., AND CASADO, M. The design and implementation of open vswitch. In *NSDI* (2015).

[44] PLOTKIN, G. D., BJØRNER, N., LOPES, N. P., RYBALCHENKO, A., AND VARGHESE, G. Scaling network verification using symmetry and surgery. In *POPL* (2016).

[45] QUAGGA. http://www.nongnu.org/quagga/.

[46] ROUTE VIEWS. http://www.routeviews.org/.

[47] SDN-IP APPLICATION. https://wiki.onosproject.org/display/ONOS/SDN-IP.

[48] SON, S., SHIN, S., YEGNESWARAN, V., PORRAS, P. A., AND GU, G. Model checking invariant security properties in OpenFlow. In *ICC* (2013).

[49] SPRING, N., MAHAJAN, R., AND WETHERALL, D. Measuring ISP topologies with Rocketfuel. In *SIGCOMM* (2002).

[50] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. SymNet: Scalable symbolic execution for modern networks. In *SIGCOMM* (2016).

[51] TAHMASBI, M. personal communication.

[52] UTTING, M., PRETSCHNER, A., AND LEGEARD, B. A taxonomy of model-based testing approaches. *Software Testing, Verification & Reliability 22*, 5 (Aug. 2012).

[53] VELNER, Y., ALPERNAS, K., PANDA, A., RABINOVICH, A., SAGIV, M., SHENKER, S., AND SHOHAM, S. Some complexity results for stateful network verification. In *TACAS* (2016).

[54] XIE, G. G., ZHANM, J., MALTZ, D. A., ZHANG, H., GREENBERG, A., HJALMTYSSON, G., AND REXFORD, J. On static reachability analysis of ip networks. In *INFOCOM* (2005).

[55] YANG, H., AND LAM, S. S. Real-time verification of network properties using atomic predicates. In *ICNP* (2013).

[56] YUAN, D., LUO, Y., ZHUANG, X., RODRIGUES, G. R., ZHAO, X., ZHANG, Y., JAIN, P. U., AND STUMM, M. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *OSDI* (2014).

[57] YUAN, L., MAI, J., SU, Z., CHEN, H., CHUAH, C.-N., AND MOHAPATRA, P. FIREMAN: A toolkit for firewall modeling and analysis. In *SP* (2006).

[58] ZENG, H., KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Automatic test packet generation. In *CoNEXT* (2012).

[59] ZENG, H., ZHANG, S., YE, F., JEYAKUMAR, V., JU, M., LIU, J., MCKEOWN, N., AND VAHDAT, A. Libra: Divide and conquer to verify forwarding tables in huge networks. In *NSDI* (2014).

A Illustration of Boolean lattice

Delta-net is based on ideas from lattice theory.[5] In particular, Delta-net leverages the concept of atoms, a form of mutually disjoint ranges that make it possible to analyze all Boolean combinations of IP prefix forwarding rules in a network. The fact that atoms induce a Boolean lattice is illustrated by the Hasse diagram [12] in Figure 9 where atoms (depicted in bold) correspond to α_0, α_1 and α_2 in Figure 5, respectively.

[5] For interested readers, a good introduction to lattice theory, whose applications in computer science are pervasive, can be found in [12]

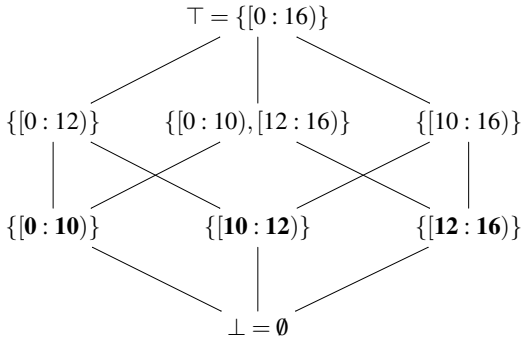


Figure 9: Boolean lattice induced by the atoms (bold) in Figure 5, assuming 4-bit numbers for simplicity.

B Proof of complexity analysis

In this appendix, we sketch the proof of the asymptotic worst-case time complexity of Algorithm 1 and 2.

Proof of Theorem 1. We analyze INSERT_RULE. Each atom split (line 2-9) requires copying the owner information from an existing atom to a newly created atom. For insertion of R rules, resulting in K atoms, this requires $O(RK)$ steps in the worst-case. In each insertion, the adjustment of labels and retrieval of the balanced binary search tree (BST) (line 12) are amortized constant-time operations per atom. Inserting each rule into the BST and finding the highest-priority rule per atom (line 14) are $O(\log M)$. By the loop (line 10-23), we get $O(RK + RK \log M) = O(RK \log M)$, concluding the proof. A similar argument proves the claim for REMOVE_RULE. □

C Comparison to previous data sets

In this appendix, we discuss how our data sets compare to previous ones used in the experimental evaluation of Veriflow [27].

In particular, it is natural to ask how our RF 1755 data set in Table 2 compares to the one used in a previous Veriflow experiment [27], which was constructed from 5 million BGP RIP entries and by 'replaying' 90,000 BGP updates. While the resulting total number of IP prefix rules in the original RF 1755 data set is not reported, the authors of the Veriflow paper note that "[t]he largest number of ECs (equivalence classes) affected by a single rule was 574; the largest verification latency was 159.2 *ms* due to an update affecting 511 ECs." For our experiments, we expect this number to be different, since we had to generate a new data set.

Running Veriflow-RI (§ 4.3.1) on our RF 1755 data set, we find that the maximum number of affected ECs on rule insertions is 319,681, which is significantly larger than the original experimental evaluation of Veriflow [27].

Data set	Memory usage (*MB*)	
	Veriflow-RI	Delta-net
Berkeley	1,089	6,208
INET	9,776	63,563
RF 1755	2,713	16,937
RF 3257	5,882	40,716
RF 6461	5,920	39,481
Airtel 1	7	61
Airtel 2	9	74
4Switch	154	785

Table 5: Memory usage of Delta-net and Veriflow-RI.

D Memory usage

In this appendix, we report the detailed memory consumption of Delta-net (§ 3) and Veriflow-RI (§ 4.3.1) using our eight data sets (§ 4.2, see Table 2).

Table 5 quantifies the memory usage of Delta-net and Veriflow-RI. In all cases, Delta-net consumes between 5 and 7 times more space than Veriflow-RI. This increase in memory consumption is offset, however, by the fact that Delta-net keeps track of the forwarding behaviour of all packets, and as a result can check properties that Veriflow-RI cannot. Nevertheless, as discussed for future work (§ 5), we are actively working on asymptotically reducing the memory consumption of Delta-net.