conference

proceedings

8th USENIX Symposium on Operating Systems Design and Implementation

San Diego, CA, USA

December 8–10, 2008

Sponsored by
The USENIX Association

USENIX®
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

in cooperation with ACM SIGOPS

For additional copies of these proceedings contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 • FAX: 510-548-5738 • office@usenix.org • http://www.usenix.org

The price is $50 for members and $60 for nonmembers.
Outside the U.S.A. and Canada, please add $20 per copy for postage (via air printed matter).

Thanks to Our Sponsors

Platinum Sponsor

Infosys®
POWERED BY INTELLECT
DRIVEN BY VALUES

Reception and Bronze Sponsors

Google™ Microsoft® Research

Gold Sponsor

National Science Foundation

Silver Sponsors

AMD hp invent vmware®

Conference Network Sponsor

Sun microsystems
The Network is the Computer™

Bronze Sponsors

ASTER intel YAHOO! RESEARCH IBM Research

Media Sponsors

ACM *Queue*
Addison-Wesley Professional/
Prentice Hall Professional/
Cisco Press

InfoSec News
ITtoolbox

Linux Journal

Linux Pro Magazine
LXer.com
StorageNetworking.org
The Register

ISBN-13: 978-1-931971-65-2

USENIX Association

Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)

December 8–10, 2008
San Diego, CA, USA

Symposium Organizers

Program Co-Chairs

Richard Draves, *Microsoft Research*
Robbert van Renesse, *Cornell University*

Program Committee

Marcos Aguilera, *Microsoft Research*
Lorenzo Alvisi, *University of Texas, Austin*
Remzi Arpaci-Dusseau, *University of Wisconsin, Madison*
Eric Brewer, *University of California, Berkeley, and Intel Research Berkeley*
Brad Chen, *Google, Inc.*
Fred Douglis, *IBM Research*
Greg Ganger, *Carnegie Mellon University*
Galen Hunt, *Microsoft Research*
Anthony Joseph, *University of California, Berkeley*
Dina Katabi, *MIT*
Kim Keeton, *HP Labs*
Idit Keidar, *Technion*
Terence Kelly, *HP Labs*
Dejan Kostić, *École Polytechnique Fédérale de Lausanne*
Philip Levis, *Stanford University*
David Lie, *University of Toronto*
Jack Lo, *VMware*
Dahlia Malkhi, *Microsoft Research*
Erich Nahum, *IBM Research*
Fernando Pedone, *University of Lugano*
Ian Pratt, *University of Cambridge*
Dave Presotto, *Google, Inc.*
Krithi Ramamritham, *IIT Bombay*
Rodrigo Rodrigues, *Max Planck Institute for Software Systems*
Wolfgang Schröder-Preikschat, *University of Erlangen-Nürnberg*
Marvin Theimer, *Amazon.com*
Leendert van Doorn, *AMD*
Geoffrey M. Voelker, *University of California, San Diego*
Jim Waldo, *Sun Microsystems, Inc.*
Helen Wang, *Microsoft Research*
Hakim Weatherspoon, *Cornell University*

Poster and Work-in-Progress (WiP) Sessions Co-Chairs

Dejan Kostić, *École Polytechnique Fédérale de Lausanne*
Philip Levis, *Stanford University*

Steering Committee

Brian Bershad, *University of Washington*
Jeff Mogul, *HP Labs*
Margo Seltzer, *Harvard University*
Ellie Young, *USENIX*

The USENIX Association Staff

External Reviewers

Atul Adya
Sharad Agarwal
Aditya Akella
Martin Arlitt
Gal Badishi
Mahesh Balakrishnan
Paul Barham
Amit Berman
Mark Bickford
Sebastian Biemueller
Hans Boehm
Bill Bolosky
Edward Bortnikov
Dhruva Chakrabarti
Anton Chernoff
Stephen Chong
Allen Clement
Olivier Crameri
Uwe Dannowski
Stephan Diestelhorst
John Douceur
Peter Druschel
Dawson Engler
Kave Eshghi
Ittai Eyal
Pedro Fonseca
Alex Friedman
Moises Goldsmidt
Albert Greenberg
Chris Grier
Saikat Guha
Ramki Gummadi
Diwaker Gupta
Maxim Gurevich
Zvika Guz
Andreas Haeberlen
Gernot Heiser
Maurice Herlihy
Michael Hicks
Michael Hohmuth
Aman Kansal
Alan Karp
Jim Larus
Charles R. Lefurgy
Harry Li
Mark Lillibridge
Roie Melamed
Alan Mislove
Tomer Morad
Brad Morrey
Robert Morris
Madanial Musuvathi
Andrew Myers
Pradeep Padala
Yoonho Park
Matt Piotrowski
Ansley Post
Tom Rodenheffer
Andrey Rybalchenko
Jim Saxe
Nicolas Schiper
Simon Schubert
David Schultz
Alex Shraer
Craig Soules
Chris Stewart
Marc Stiegler
Ram Swaminathan
Chunqiang Tang
Doug Terry
Nedeljko Vasic
Luis Veiga
Michael Vrable
Ollie Williams
Ted Wobber
Yinglian Xie
Praveen Yalagandula
Junfeng Yang
Jian Yin
Ken Yocum
Fang Yu
Ming Zhang

8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)
December 8–10, 2008
San Diego, CA, USA

Index of Authors vi
Message from the Program Co-Chairs vii

Monday, December 8

Cloud Computing

DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language . . 1
Yuan Yu, Michael Isard, Dennis Fetterly, and Mihai Budiu, Microsoft Research Silicon Valley; Úlfar Erlingsson, Reykjavík University, Iceland, and Microsoft Research Silicon Valley; Pradeep Kumar Gunda and Jon Currey, Microsoft Research Silicon Valley

Everest: Scaling Down Peak Loads Through I/O Off-Loading 15
Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony Rowstron, Microsoft Research Cambridge, United Kingdom

Improving MapReduce Performance in Heterogeneous Environments 29
Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica, University of California, Berkeley

OS Architecture

Corey: An Operating System for Many Cores 43
Silas Boyd-Wickizer, Massachusetts Institute of Technology; Haibo Chen, Rong Chen, and Yandong Mao, Fudan University; Frans Kaashoek, Robert Morris, and Aleksey Pesterev, Massachusetts Institute of Technology; Lex Stein and Ming Wu, Microsoft Research Asia; Yuehua Dai, Xi'an Jiaotong University; Yang Zhang, Massachusetts Institute of Technology; Zheng Zhang, Microsoft Research Asia

CuriOS: Improving Reliability through Operating System Structure 59
Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell, University of Illinois at Urbana-Champaign

Redline: First Class Support for Interactivity in Commodity Operating Systems 73
Ting Yang, Tongping Liu, and Emery D. Berger, University of Massachusetts Amherst; Scott F. Kaplan, Amherst College; J. Eliot B. Moss, University of Massachusetts Amherst

Monitoring

Network Imprecision: A New Consistency Metric for Scalable Monitoring 87
Navendu Jain, Microsoft Research; Prince Mahajan and Dmitry Kit, University of Texas at Austin; Praveen Yalagandula, HP Labs; Mike Dahlin and Yin Zhang, University of Texas at Austin

Lightweight, High-Resolution Monitoring for Troubleshooting Production Systems 103
Sapan Bhatia, Princeton University; Abhishek Kumar, Google Inc.; Marc E. Fiuczynski and Larry Peterson, Princeton University

Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions 117
Xu Chen, University of Michigan; Ming Zhang, Microsoft Research; Z. Morley Mao, University of Michigan; Paramvir Bahl, Microsoft Research

Tuesday, December 9

File Systems

SQCK: A Declarative File System Checker 131
Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin, Madison

Transactional Flash 147
Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou, Microsoft Research, Silicon Valley

Avoiding File System Micromanagement with Range Writes 161
Ashok Anand and Sayandeep Sen, University of Wisconsin, Madison; Andrew Krioukov, University of California, Berkeley; Florentina Popovici, Google; Aditya Akella, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Suman Banerjee, University of Wisconsin, Madison

Programming Language Techniques

Binary Translation Using Peephole Superoptimizers 177
Sorav Bansal and Alex Aiken, Stanford University

R2: An Application-Level Kernel for Record and Replay 193
Zhenyu Guo, Microsoft Research Asia; Xi Wang, Tsinghua University; Jian Tang and Xuezheng Liu, Microsoft Research Asia; Zhilei Xu, Tsinghua University; Ming Wu, Microsoft Research Asia; M. Frans Kaashoek, MIT CSAIL; Zheng Zhang, Microsoft Research Asia

KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs 209
Cristian Cadar, Daniel Dunbar, and Dawson Engler, Stanford University

Security

Hardware Enforcement of Application Security Policies Using Tagged Memory 225
Nickolai Zeldovich, Massachusetts Institute of Technology; Hari Kannan, Michael Dalton, and Christos Kozyrakis, Stanford University

Device Driver Safety Through a Reference Validation Mechanism 241
Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider, Cornell University

Digging for Data Structures 255
Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King, University of Illinois at Urbana-Champaign

Dealing with Concurrency Bugs

Finding and Reproducing Heisenbugs in Concurrent Programs 267
Madanlal Musuvathi, Shaz Qadeer, and Thomas Ball, Microsoft Research; Gerard Basler, ETH Zurich; Piramanayagam Arumuga Nainar, University of Wisconsin, Madison; Iulian Neamtiu, University of California, Riverside

Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs 281
Yin Wang, University of Michigan and Hewlett-Packard Laboratories; Terence Kelly, Hewlett-Packard Laboratories; Manjunath Kudlur, Stéphane Lafortune, and Scott Mahlke, University of Michigan

Deadlock Immunity: Enabling Systems to Defend Against Deadlocks 295
Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Wednesday, December 10

Various Good Things

Difference Engine: Harnessing Memory Redundancy in Virtual Machines 309
Diwaker Gupta, University of California, San Diego; Sangmin Lee, University of Texas at Austin; Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat, University of California, San Diego

Quanto: Tracking Energy in Networked Embedded Systems 323
Rodrigo Fonseca, University of California, Berkeley, and Yahoo! Research; Prabal Dutta, University of California, Berkeley; Philip Levis, Stanford University; Ion Stoica, University of California, Berkeley

Leveraging Legacy Code to Deploy Desktop Applications on the Web 339
John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch, Microsoft Research

Wide-Area Distributed Systems

FlightPath: Obedience vs. Choice in Cooperative Services. 355
Harry C. Li and Allen Clement, University of Texas at Austin; Mirco Marchetti, University of Modena and Reggio Emilia; Manos Kapritsos, Luke Robison, Lorenzo Alvisi, and Mike Dahlin, University of Texas at Austin

Mencius: Building Efficient Replicated State Machine for WANs 369
Yanhua Mao, CSE, University of California, San Diego; Flavio P. Junqueira, Yahoo! Research Barcelona; Keith Marzullo, CSE, University of California, San Diego

Index of Authors

Aiken, Alex 177
Akella, Aditya 161
Alvisi, Lorenzo 355
Anand, Ashok 161
Arpaci-Dusseau, Andrea C. 131, 161
Arpaci-Dusseau, Remzi H. 131, 161
Bahl, Paramvir 117
Ball, Thomas 267
Banerjee, Suman 161
Bansal, Sorav 177
Basler, Gerard 267
Berger, Emery D. 73
Bhatia, Sapan 103
Boyd-Wickizer, Silas 43
Budiu, Mihai 1
Cadar, Cristian 209
Campbell, Roy H. 59
Candea, George 295
Carlyle, Jeffrey C. 59
Chan, Ellick M. 59
Chen, Haibo 43
Chen, Rong 43
Chen, Xu 117
Clement, Allen 355
Cozzie, Anthony 255
Currey, Jon 1
Dahlin, Mike 87, 355
Dai, Yuehua 43
Dalton, Michael 225
David, Francis M. 59
Donnelly, Austin 15
Douceur, John R. 339
Dunbar, Daniel 209
Dutta, Prabal 323
Elnikety, Sameh 15
Elson, Jeremy 339
Engler, Dawson 209
Erlingsson, Úlfar 1
Fetterly, Dennis 1
Fiuczynski, Marc E. 103
Fonseca, Rodrigo 323
Gunawi, Haryadi S. 131
Gunda, Pradeep Kumar 1
Guo, Zhenyu 193
Gupta, Diwaker 309
Howell, Jon 339
Isard, Michael 1
Jain, Navendu 87
Joseph, Anthony D. 29
Jula, Horatiu 295
Junqueira, Flavio P. 369
Kaashoek, M. Frans 43, 193
Kannan, Hari 225
Kaplan, Scott F. 73
Kapritsos, Manos 355
Katz, Randy 29
Kelly, Terence 281
King, Samuel T. 255
Kit, Dmitry 87
Konwinski, Andy 29
Kozyrakis, Christos 225
Krioukov, Andrew 161
Kudlur, Manjunath 281
Kumar, Abhishek 103
Lafortune, Stéphane 281
Lee, Sangmin 309
Levis, Philip 323
Li, Harry C. 355
Liu, Tongping 73
Liu, Xuezheng 193
Lorch, Jacob R. 339
Mahajan, Prince 87
Mahlke, Scott 281
Mao, Yandong 43
Mao, Yanhua 369
Mao, Z. Morley 117
Marchetti, Mirco 355
Marzullo, Keith 369
Morris, Robert 43
Moss, J. Eliot B. 73
Musuvathi, Madanlal 267
Nainar, Piramanayagam Arumuga 267
Narayanan, Dushyanth 15
Neamtiu, Iulian 267
Pesterev, Aleksey 43
Peterson,, Larry 103
Popovici, Florentina 161
Prabhakaran, Vijayan 147
Qadeer, Shaz 267
Rajimwale, Abhishek 131
Reynolds, Patrick 241
Robison, Luke 355
Rodeheffer, Thomas L. 147
Rowstron, Antony 15
Savage, Stefan 309
Schneider, Fred B. 241
Sen, Sayandeep 161
Sirer, Emin Gün 241
Snoeren, Alex C. 309
Stein, Lex 43
Stoica, Ion 29, 323
Stratton, Frank 255
Tang, Jian 193
Thereska, Eno 15
Tralamazza, Daniel 295
Vahdat, Amin 309
Varghese, George 309
Voelker, Geoffrey M. 309
Vrable, Michael 309
Walsh, Kevin 241
Wang, Xi 193
Wang, Yin 281
Williams, Dan 241
Wu, Ming 43, 193
Xu, Zhilei 193
Xue, Hui 255
Yalagandula, Praveen 87
Yang, Ting 73
Yu, Yuan 1
Zaharia, Matei 29
Zamfir, Cristian 295
Zeldovich, Nickolai 225
Zhang, Ming 117
Zhang, Yang 43
Zhang, Yin 87
Zhang, Zheng 43, 193
Zhou, Lidong 147

Message from the Program Co-Chairs

Dear colleagues,

We are very saddened that Jay Lepreau, OSDI's founder and first program chair in 1994, passed away on September 15 this year. Although our community will miss him deeply, we are proud to carry on his vision of OSDI as a top-tier systems conference and complement to SOSP for bringing together professionals from academic and industrial backgrounds.

OSDI '08 received 193 submissions, tying the record of four years ago. We used Eddie Kohler's HotCRP software to handle submissions and Geoff Voelker's Banal program in the submissions process to check compliance with the formatting guidelines. We also used the commercial word-counting program AnyCount to check for papers that somehow managed to squeeze an excessive amount of content into 14 pages. (Most submissions had well over 10,000 words.) One submission stood out in violation of the guidelines, but as this was due to a misunderstanding, we allowed the authors to delete some subsections. Ultimately, that paper was not accepted.

Following the lead of several recent systems and networking conferences, we used a two-tiered PC structure to spread the reviewing load and three reviewing rounds to focus effort on the most promising submissions. We had 16 "heavy-load" PC members (including the co-chairs) who agreed to participate in all three reviewing rounds and attend the PC meeting. The 17 "light-load" PC members participated in the first two reviewing rounds. We also solicited external reviews for papers for which additional expertise was needed.

In the first round, all papers received three reviews. With few exceptions, the 105 papers receiving at least one "accept" recommendation or at least two "weak accept" recommendations moved forward to the second round. In the second round, surviving papers received two more reviews. The 65 papers with at least one "accept" recommendation moved forward to the third round. In this last round, surviving papers received at least two more reviews. Ultimately, papers discussed at the PC meeting had at least seven reviews, with at least four reviews by heavy-load members. In all, the 193 submitted papers received 948 reviews.

At the PC meeting, we relied on HotCRP to manage the discussion and record the committee's decisions. Because HotCRP prevents PC members from viewing papers for which they are conflicted, this meant that during the meeting and until the official notifications went out, the PC did not know the outcome for conflicted papers, including their own.

We accepted 26 of the 193 papers, or 13.5%. The PC took a broad view of the systems area and ultimately accepted papers on a diverse set of topics, including operating systems (of course), cloud computing, file systems, distributed systems, security, concurrency, and systems based on techniques from the PL community. The PC also continued the OSDI tradition of preferring "exciting but flawed" submissions over "boring but correct" ones. As usual, we assigned each accepted paper a shepherd to supervise the revision process. We believe the final program represents some of the best work being done in systems research today.

Creating a conference program is a significant undertaking. Our role as program co-chairs was to lead and give direction to the many folks who did a tremendous amount of hard work. We thank the authors for their efforts, especially those whose papers we could not accept. We thank the program committee for their hard work—heavy-load members reviewed at least 33 papers each and light-load members reviewed at least 20 papers each. We thank our external reviewers for contributing much-needed reviews on short notice. We thank the steering committee for its support and guidance. We thank Eddie and Geoff for developing and supporting their software. Finally, we thank the USENIX staff for all the behind-the-scenes work that makes the conference actually happen.

Richard Draves, Microsoft Research
Robbert van Renesse, Cornell University
Program Co-Chairs

DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language

Yuan Yu Michael Isard Dennis Fetterly Mihai Budiu
Úlfar Erlingsson[1] Pradeep Kumar Gunda Jon Currey
Microsoft Research Silicon Valley [1]*joint affiliation, Reykjavík University, Iceland*

Abstract

DryadLINQ is a system and a set of language extensions that enable a new programming model for large scale distributed computing. It generalizes previous execution environments such as SQL, MapReduce, and Dryad in two ways: by adopting an expressive data model of strongly typed .NET objects; and by supporting general-purpose imperative and declarative operations on datasets within a traditional high-level programming language.

A DryadLINQ program is a sequential program composed of LINQ expressions performing arbitrary side-effect-free transformations on datasets, and can be written and debugged using standard .NET development tools. The DryadLINQ system automatically and transparently translates the data-parallel portions of the program into a distributed execution plan which is passed to the Dryad execution platform. Dryad, which has been in continuous operation for several years on production clusters made up of thousands of computers, ensures efficient, reliable execution of this plan.

We describe the implementation of the DryadLINQ compiler and runtime. We evaluate DryadLINQ on a varied set of programs drawn from domains such as web-graph analysis, large-scale log mining, and machine learning. We show that excellent absolute performance can be attained—a general-purpose sort of 10^{12} Bytes of data executes in 319 seconds on a 240-computer, 960-disk cluster—as well as demonstrating near-linear scaling of execution time on representative applications as we vary the number of computers used for a job.

1 Introduction

The DryadLINQ system is designed to make it easy for a wide variety of developers to compute effectively on large amounts of data. DryadLINQ programs are written as imperative or declarative operations on datasets within a traditional high-level programming language, using an expressive data model of strongly typed .NET objects. The main contribution of this paper is a set of language extensions and a corresponding system that can automatically and transparently compile imperative programs in a general-purpose language into distributed computations that execute efficiently on large computing clusters.

Our goal is to give the programmer the illusion of writing for a single computer and to have the system deal with the complexities that arise from scheduling, distribution, and fault-tolerance. Achieving this goal requires a wide variety of components to interact, including cluster-management software, distributed-execution middleware, language constructs, and development tools. Traditional parallel databases (which we survey in Section 6.1) as well as more recent data-processing systems such as MapReduce [15] and Dryad [26] demonstrate that it is possible to implement high-performance large-scale execution engines at modest financial cost, and clusters running such platforms are proliferating. Even so, their programming interfaces all leave room for improvement. We therefore believe that the language issues addressed in this paper are currently among the most pressing research areas for data-intensive computing, and our work on the DryadLINQ system stems from this belief.

DryadLINQ exploits LINQ (Language INtegrated Query [2], a set of .NET constructs for programming with datasets) to provide a powerful hybrid of declarative and imperative programming. The system is designed to provide flexible and efficient distributed computation in any LINQ-enabled programming language including C#, VB, and F#. Objects in DryadLINQ datasets can be of any .NET type, making it easy to compute with data such as image patches, vectors, and matrices. DryadLINQ programs can use traditional structuring constructs such as functions, modules, and libraries, and express iteration using standard loops. Crucially, the distributed execution layer employs a fully functional, declarative description of the data-parallel component of the computation,

which enables sophisticated rewritings and optimizations like those traditionally employed by parallel databases.

In contrast, parallel databases implement only declarative variants of SQL queries. There is by now a widespread belief that SQL is too limited for many applications [15, 26, 31, 34, 35]. One problem is that, in order to support database requirements such as in-place updates and efficient transactions, SQL adopts a very restrictive type system. In addition, the declarative "query-oriented" nature of SQL makes it difficult to express common programming patterns such as iteration [14]. Together, these make SQL unsuitable for tasks such as machine learning, content parsing, and web-graph analysis that increasingly must be run on very large datasets.

The MapReduce system [15] adopted a radically simplified programming abstraction, however even common operations like database Join are tricky to implement in this model. Moreover, it is necessary to embed MapReduce computations in a scripting language in order to execute programs that require more than one reduction or sorting stage. Each MapReduce instantiation is self-contained and no automatic optimizations take place across their boundaries. In addition, the lack of any type-system support or integration between the MapReduce stages requires programmers to explicitly keep track of objects passed between these stages, and may complicate long-term maintenance and re-use of software components.

Several domain-specific languages have appeared on top of the MapReduce abstraction to hide some of this complexity from the programmer, including Sawzall [32], Pig [31], and other unpublished systems such as Facebook's HIVE. These offer a limited hybridization of declarative and imperative programs and generalize SQL's stored-procedure model. Some whole-query optimizations are automatically applied by these systems across MapReduce computation boundaries. However, these approaches inherit many of SQL's disadvantages, adopting simple custom type systems and providing limited support for iterative computations. Their support for optimizations is less advanced than that in DryadLINQ, partly because the underlying MapReduce execution platform is much less flexible than Dryad.

DryadLINQ and systems such as MapReduce are also distinguished from traditional databases [25] by having *virtualized* expression plans. The planner allocates resources independent of the actual cluster used for execution. This means both that DryadLINQ can run plans requiring many more steps than the instantaneously-available computation resources would permit, and that the computational resources can change dynamically, e.g. due to faults—in essence, we have an extra degree of freedom in buffer management compared with traditional schemes [21, 24, 27, 28, 29]. A downside of virtualization is that it requires intermediate results to be stored to persistent media, potentially increasing computation latency.

This paper makes the following contributions to the literature:

- We have demonstrated a new hybrid of declarative and imperative programming, suitable for large-scale data-parallel computing using a rich object-oriented programming language.
- We have implemented the DryadLINQ system and validated the hypothesis that DryadLINQ programs can be automatically optimized and efficiently executed on large clusters.
- We have designed a small set of operators that improve LINQ's support for coarse-grain parallelization while preserving its programming model.

Section 2 provides a high-level overview of the steps involved when a DryadLINQ program is run. Section 3 discusses LINQ and the extensions to its programming model that comprise DryadLINQ along with simple illustrative examples. Section 4 describes the DryadLINQ implementation and its interaction with the low-level Dryad primitives. In Section 5 we evaluate our system using several example applications at a variety of scales. Section 6 compares DryadLINQ to related work and Section 7 discusses limitations of the system and lessons learned from its development.

2 System Architecture

DryadLINQ compiles LINQ programs into distributed computations running on the Dryad cluster-computing infrastructure [26]. A Dryad job is a directed acyclic graph where each vertex is a program and edges represent data channels. At run time, vertices are processes communicating with each other through the channels, and each channel is used to transport a finite sequence of data records. The data model and serialization are provided by higher-level software layers, in this case DryadLINQ.

Figure 1 illustrates the Dryad system architecture. The execution of a Dryad job is orchestrated by a centralized "job manager." The job manager is responsible for: (1) instantiating a job's dataflow graph; (2) scheduling processes on cluster computers; (3) providing fault-tolerance by re-executing failed or slow processes; (4) monitoring the job and collecting statistics; and (5) transforming the job graph dynamically according to user-supplied policies.

A cluster is typically controlled by a task scheduler, separate from Dryad, which manages a batch queue of jobs and executes a few at a time subject to cluster policy.

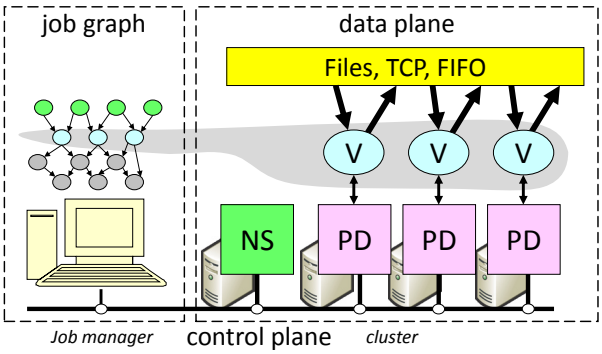


Figure 1: *Dryad system architecture. NS is the name server which maintains the cluster membership. The job manager is responsible for spawning vertices (V) on available computers with the help of a remote-execution and monitoring daemon (PD). Vertices exchange data through files, TCP pipes, or shared-memory channels. The grey shape indicates the vertices in the job that are currently running and the correspondence with the job execution graph.*

2.1 DryadLINQ Execution Overview

Figure 2 shows the flow of execution when a program is executed by DryadLINQ.

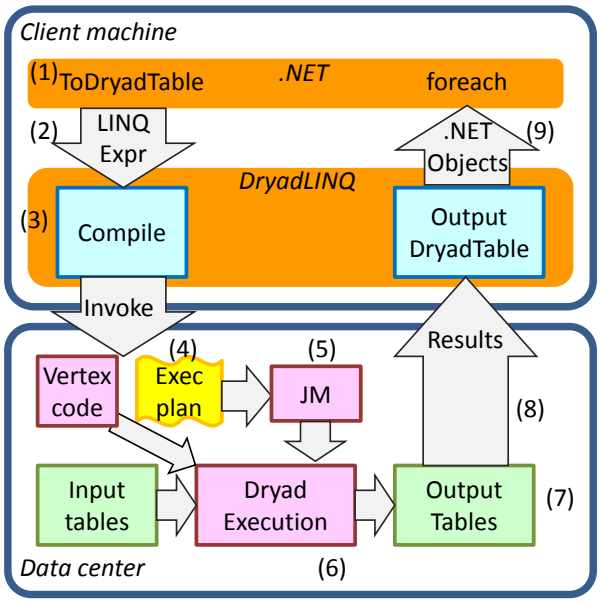


Figure 2: *LINQ-expression execution in DryadLINQ.*

Step 1. A .NET user application runs. It creates a DryadLINQ expression object. Because of LINQ's deferred evaluation (described in Section 3), the actual execution of the expression has not occurred.

Step 2. The application calls `ToDryadTable` triggering a data-parallel execution. The expression object is handed to DryadLINQ.

Step 3. DryadLINQ compiles the LINQ expression into a distributed Dryad execution plan. It performs: (a) the decomposition of the expression into subexpressions, each to be run in a separate Dryad vertex; (b) the generation of code and static data for the remote Dryad vertices; and (c) the generation of serialization code for the required data types. Section 4 describes these steps in detail.

Step 4. DryadLINQ invokes a custom, DryadLINQ-specific, Dryad job manager. The job manager may be executed behind a cluster firewall.

Step 5. The job manager creates the job graph using the plan created in Step 3. It schedules and spawns the vertices as resources become available. See Figure 1.

Step 6. Each Dryad vertex executes a vertex-specific program (created in Step 3b).

Step 7. When the Dryad job completes successfully it writes the data to the output table(s).

Step 8. The job manager process terminates, and it returns control back to DryadLINQ. DryadLINQ creates the local `DryadTable` objects encapsulating the outputs of the execution. These objects may be used as inputs to subsequent expressions in the user program. Data objects within a `DryadTable` output are fetched to the local context only if explicitly dereferenced.

Step 9. Control returns to the user application. The iterator interface over a `DryadTable` allows the user to read its contents as .NET objects.

Step 10. The application may generate subsequent DryadLINQ expressions, to be executed by a repetition of Steps 2–9.

3 Programming with DryadLINQ

In this section we highlight some particularly useful and distinctive aspects of DryadLINQ. More details on the programming model may be found in LINQ language reference [2] and materials on the DryadLINQ project website [1] including a language tutorial. A companion technical report [38] contains listings of some of the sample programs described below.

3.1 LINQ

The term LINQ [2] refers to a set of .NET constructs for manipulating sets and sequences of data items. We describe it here as it applies to C# but DryadLINQ programs have been written in other .NET languages including F#. The power and extensibility of LINQ derive from a set of design choices that allow the programmer to express complex computations over datasets while giving the runtime great leeway to decide how these computations should be implemented.

The base type for a LINQ collection is `IEnumerable<T>`. From a programmer's perspective, this is

an abstract dataset of objects of type T that is accessed using an iterator interface. LINQ also defines the IQueryable<T> interface which is a subtype of IEnumerable<T> and represents an (unevaluated) expression constructed by combining LINQ datasets using LINQ operators. We need make only two observations about these types: (a) in general the programmer neither knows nor cares what concrete type implements any given dataset's IEnumerable interface; and (b) DryadLINQ composes all LINQ expressions into IQueryable objects and defers evaluation until the result is needed, at which point the expression graph within the IQueryable is optimized and executed in its entirety on the cluster. Any IQueryable object can be used as an argument to multiple operators, allowing efficient re-use of common subexpressions.

LINQ expressions are statically strongly typed through use of nested generics, although the compiler hides much of the type-complexity from the user by providing a range of "syntactic sugar." Figure 3 illustrates LINQ's syntax with a fragment of a simple example program that computes the top-ranked results for each query in a stored corpus. Two versions of the same LINQ expression are shown, one using a declarative SQL-like syntax, and the second using the object-oriented style we adopt for more complex programs.

The program first performs a Join to "look up" the static rank of each document contained in a scoreTriples tuple and then computes a new rank for that tuple, combining the query-dependent score with the static score inside the constructor for QueryScoreDocIDTriple. The program next groups the resulting tuples by query, and outputs the top-ranked results for each query. The full example program is included in [38].

The second, object-oriented, version of the example illustrates LINQ's use of C#'s lambda expressions. The Join method, for example, takes as arguments a dataset to perform the Join against (in this case staticRank) and three functions. The first two functions describe how to determine the keys that should be used in the Join. The third function describes the Join function itself. Note that the compiler performs static type inference to determine the concrete types of var objects and anonymous lambda expressions so the programmer need not remember (or even know) the type signatures of many subexpressions or helper functions.

```
// SQL-style syntax to join two input sets:
// scoreTriples and staticRank
var adjustedScoreTriples =
  from d in scoreTriples
  join r in staticRank on d.docID equals r.key
  select new QueryScoreDocIDTriple(d, r);
var rankedQueries =
  from s in adjustedScoreTriples
  group s by s.query into g
  select TakeTopQueryResults(g);

// Object-oriented syntax for the above join
var adjustedScoreTriples =
  scoreTriples.Join(staticRank,
    d => d.docID, r => r.key,
    (d, r) => new QueryScoreDocIDTriple(d, r));
var groupedQueries =
  adjustedScoreTriples.GroupBy(s => s.query);
var rankedQueries =
  groupedQueries.Select(
    g => TakeTopQueryResults(g));
```

Figure 3: *A program fragment illustrating two ways of expressing the same operation. The first uses LINQ's declarative syntax, and the second uses object-oriented interfaces. Statements such as* r => r.key *use C#'s syntax for anonymous lambda expressions.*

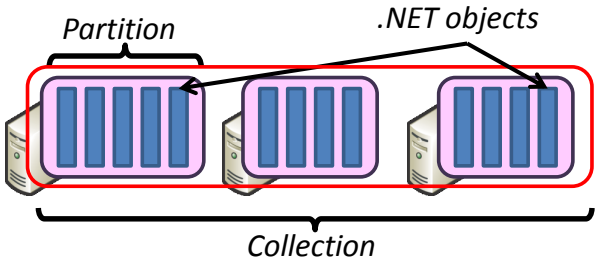


Figure 4: *The DryadLINQ data model: strongly-typed collections of .NET objects partitioned on a set of computers.*

3.2 DryadLINQ Constructs

DryadLINQ preserves the LINQ programming model and extends it to data-parallel programming by defining a small set of new operators and datatypes.

The DryadLINQ data model is a distributed implementation of LINQ collections. Datasets may still contain arbitrary .NET types, but each DryadLINQ dataset is in general distributed across the computers of a cluster, partitioned into disjoint pieces as shown in Figure 4. The partitioning strategies used—hash-partitioning, range-partitioning, and round-robin—are familiar from parallel databases [18]. This dataset partitioning is managed transparently by the system unless the programmer explicitly overrides the optimizer's choices.

The inputs and outputs of a DryadLINQ computation are represented by objects of type DryadTable<T>, which is a subtype of IQueryable<T>. Subtypes of DryadTable<T> support underlying storage providers that include distributed filesystems, collections of NTFS files, and sets of SQL tables. DryadTable objects may include metadata read from the file system describing table properties such as schemas for the data items contained in the table, and partitioning schemes which the DryadLINQ optimizer can use to generate more efficient executions. These optimizations, along with issues such

as data serialization and compression, are discussed in Section 4.

The primary restriction imposed by the DryadLINQ system to allow distributed execution is that *all the functions called in DryadLINQ expressions must be side-effect free*. Shared objects can be referenced and read freely and will be automatically serialized and distributed where necessary. However, if any shared object is modified, the result of the computation is undefined. DryadLINQ does not currently check or enforce the absence of side-effects.

The inputs and outputs of a DryadLINQ computation are specified using the GetTable<T> and ToDryadTable<T> operators, e.g.:

```
var input = GetTable<LineRecord>("file://in.tbl");
var result = MainProgram(input, ...);
var output = ToDryadTable(result, "file://out.tbl");
```

Tables are referenced by URI strings that indicate the storage system to use as well as the name of the partitioned dataset. Variants of ToDryadTable can simultaneously invoke multiple expressions and generate multiple output DryadTables in a single distributed Dryad job. This feature (also encountered in parallel databases such as Teradata) can be used to avoid recomputing or materializing common subexpressions.

DryadLINQ offers two data re-partitioning operators: HashPartition<T,K> and RangePartition<T,K>. These operators are needed to enforce a partitioning on an output dataset and they may also be used to override the optimizer's choice of execution plan. From a LINQ perspective, however, they are no-ops since they just reorganize a collection without changing its contents. The system allows the implementation of additional re-partitioning operators, but we have found these two to be sufficient for a wide class of applications.

The remaining new operators are Apply and Fork, which can be thought of as an "escape-hatch" that a programmer can use when a computation is needed that cannot be expressed using any of LINQ's built-in operators. Apply takes a function f and passes to it an iterator over the entire input collection, allowing arbitrary streaming computations. As a simple example, Apply can be used to perform "windowed" computations on a sequence, where the ith entry of the output sequence is a function on the range of input values $[i, i + d]$ for a fixed window of length d. The applications of Apply are much more general than this and we discuss them further in Section 7. The Fork operator is very similar to Apply except that it takes a single input and generates multiple output datasets. This is useful as a performance optimization to eliminate common subcomputations, e.g. to implement a document parser that outputs both plain text and a bibliographic entry to separate tables.

If the DryadLINQ system has no further information about f, Apply (or Fork) will cause all of the computation to be serialized onto a single computer. More often, however, the user supplies annotations on f that indicate conditions under which Apply can be parallelized. The details are too complex to be described in the space available, but quite general "conditional homomorphism" is supported—this means that the application can specify conditions on the partitioning of a dataset under which Apply can be run independently on each partition. DryadLINQ will automatically re-partition the data to match the conditions if necessary.

DryadLINQ allows programmers to specify annotations of various kinds. These provide manual hints to guide optimizations that the system is unable to perform automatically, while preserving the semantics of the program. As mentioned above, the Apply operator makes use of annotations, supplied as simple .NET attributes, to indicate opportunities for parallelization. There are also Resource annotations to discriminate functions that require constant storage from those whose storage grows along with the input collection size—these are used by the optimizer to determine buffering strategies and decide when to pipeline operators in the same process. The programmer may also declare that a dataset has a particular partitioning scheme if the file system does not store sufficient metadata to determine this automatically.

The DryadLINQ optimizer produces good automatic execution plans for most programs composed of standard LINQ operators, and annotations are seldom needed unless an application uses complex Apply statements.

3.3 Building on DryadLINQ

Many programs can be directly written using the DryadLINQ primitives. Nevertheless, we have begun to build libraries of common subroutines for various application domains. The ease of defining and maintaining such libraries using C#'s functions and interfaces highlights the advantages of embedding data-parallel constructs within a high-level language.

The MapReduce programming model from [15] can be compactly stated as follows (eliding the precise type signatures for clarity):

```
public static MapReduce( // returns set of Rs
    source, // set of Ts
    mapper, // function from T → Ms
    keySelector, // function from M → K
    reducer // function from (K,Ms) → Rs
) {
    var mapped = source.SelectMany(mapper);
    var groups = mapped.GroupBy(keySelector);
    return groups.SelectMany(reducer);
}
```

Section 4 discusses the execution plan that is automatically generated for such a computation by the DryadLINQ optimizer.

We built a general-purpose library for manipulating numerical data to use as a platform for implementing machine-learning algorithms, some of which are described in Section 5. The applications are written as traditional programs calling into library functions, and make no explicit reference to the distributed nature of the computation. Several of these algorithms need to iterate over a data transformation until convergence. In a traditional database this would require support for recursive expressions, which are tricky to implement [14]; in DryadLINQ it is trivial to use a C# loop to express the iteration. The companion technical report [38] contains annotated source for some of these algorithms.

4 System Implementation

This section describes the DryadLINQ parallelizing compiler. We focus on the generation, optimization, and execution of the distributed execution plan, corresponding to step 3 in Figure 2. The DryadLINQ optimizer is similar in many respects to classical database optimizers [25]. It has a static component, which generates an execution plan, and a dynamic component, which uses Dryad policy plug-ins to optimize the graph at run time.

4.1 Execution Plan Graph

When it receives control, DryadLINQ starts by converting the raw LINQ expression into an execution plan graph (EPG), where each node is an operator and edges represent its inputs and outputs. The EPG is closely related to a traditional database query plan, but we use the more general terminology of execution plan to encompass computations that are not easily formulated as "queries." The EPG is a directed acyclic graph—the existence of common subexpressions and operators like `Fork` means that EPGs cannot always be described by trees. DryadLINQ then applies term-rewriting optimizations on the EPG. The EPG is a "skeleton" of the Dryad data-flow graph that will be executed, and each EPG node is replicated at run time to generate a Dryad "stage" (a collection of vertices running the same computation on different partitions of a dataset). The optimizer annotates the EPG with metadata properties. For edges, these include the .NET type of the data and the compression scheme, if any, used after serialization. For nodes, they include details of the partitioning scheme used, and ordering information within each partition. The output of a node, for example, might be a dataset that is hash-partitioned by a particular key, and sorted according to that key within each partition; this information can be used by subsequent `OrderBy` nodes to choose an appropriate distributed sort algorithm as described below in Section 4.2.3. The properties are seeded from the LINQ expression tree and the input and output tables' metadata, and propagated and updated during EPG rewriting.

Propagating these properties is substantially harder in the context of DryadLINQ than for a traditional database. The difficulties stem from the much richer data model and expression language. Consider one of the simplest operations: `input.Select(x => f(x))`. If `f` is a simple expression, e.g. `x.name`, then it is straightforward for DryadLINQ to determine which properties can be propagated. However, for arbitrary `f` it is in general impossible to determine whether this transformation preserves the partitioning properties of the input.

Fortunately, DryadLINQ can usually infer properties in the programs typical users write. Partition and sort key properties are stored as expressions, and it is often feasible to compare these for equality using a combination of static typing, static analysis, and reflection. The system also provides a simple mechanism that allows users to assert properties of an expression when they cannot be determined automatically.

4.2 DryadLINQ Optimizations

DryadLINQ performs both static and dynamic optimizations. The static optimizations are currently greedy heuristics, although in the future we may implement cost-based optimizations as used in traditional databases. The dynamic optimizations are applied during Dryad job execution, and consist in rewriting the job graph depending on run-time data statistics. Our optimizations are sound in that a failure to compute properties simply results in an inefficient, though correct, execution plan.

4.2.1 Static Optimizations

DryadLINQ's static optimizations are conditional graph rewriting rules triggered by a predicate on EPG node properties. Most of the static optimizations are focused on minimizing disk and network I/O. The most important are:

Pipelining: Multiple operators may be executed in a single process. The pipelined processes are themselves LINQ expressions and can be executed by an existing single-computer LINQ implementation.

Removing redundancy: DryadLINQ removes unnecessary hash- or range-partitioning steps.

Eager Aggregation: Since re-partitioning datasets is expensive, down-stream aggregations are moved in front of partitioning operators where possible.

I/O reduction: Where possible, DryadLINQ uses Dryad's TCP-pipe and in-memory FIFO channels instead of persisting temporary data to files. The system by default compresses data before performing a partitioning, to reduce network traffic. Users can manually override compression settings to balance CPU usage with network load if the optimizer makes a poor decision.

4.2.2 Dynamic Optimizations

DryadLINQ makes use of hooks in the Dryad API to dynamically mutate the execution graph as information from the running job becomes available. Aggregation gives a major opportunity for I/O reduction since it can be optimized into a tree according to locality, aggregating data first at the computer level, next at the rack level, and finally at the cluster level. The topology of such an aggregation tree can only be computed at run time, since it is dependent on the dynamic scheduling decisions which allocate vertices to computers. DryadLINQ automatically uses the dynamic-aggregation logic present in Dryad [26].

Dynamic data partitioning sets the number of vertices in each stage (i.e., the number of partitions of each dataset) at run time based on the size of its input data. Traditional databases usually estimate dataset sizes statically, but these estimates can be very inaccurate, for example in the presence of correlated queries. DryadLINQ supports dynamic hash and range partitions—for range partitions both the number of partitions and the partitioning key ranges are determined at run time by sampling the input dataset.

4.2.3 Optimizations for `OrderBy`

DryadLINQ's logic for sorting a dataset `d` illustrates many of the static and dynamic optimizations available to the system. Different strategies are adopted depending on `d`'s initial partitioning and ordering. Figure 5 shows the evolution of an `OrderBy` node `O` in the most complex case, where `d` is not already range-partitioned by the correct sort key, nor are its partitions individually ordered by the key. First, the dataset must be re-partitioned. The `DS` stage performs deterministic sampling of the input dataset. The samples are aggregated by a histogram vertex `H`, which determines the partition keys as a function of data distribution (load-balancing the computation in the next stage). The `D` vertices perform the actual repartitioning, based on the key ranges computed by `H`. Next, a merge node `M` interleaves the inputs, and a `S` node sorts them. `M` and `S` are pipelined in a single process, and communicate using iterators. The number of partitions in the `DS+H+D` stage is chosen at run time

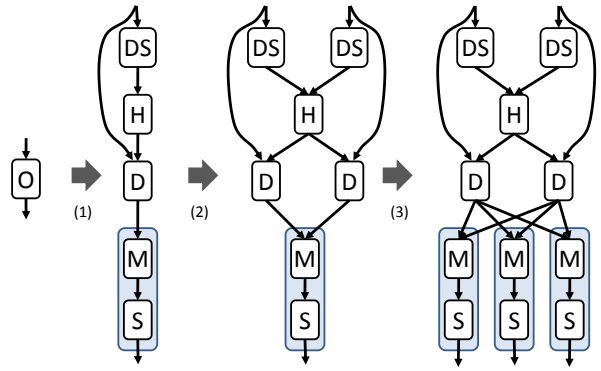


Figure 5: *Distributed sort optimization described in Section 4.2.3. Transformation (1) is static, while (2) and (3) are dynamic.*

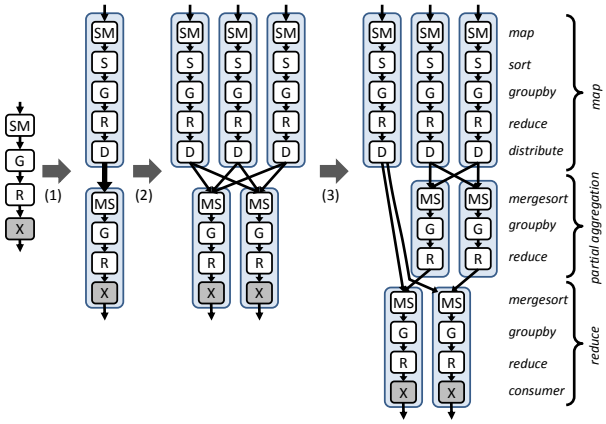


Figure 6: *Execution plan for MapReduce, described in Section 4.2.4. Step (1) is static, (2) and (3) are dynamic based on the volume and location of the data in the inputs.*

based on the number of partitions in the preceding computation, and the number of partitions in the `M+S` stage is chosen based on the volume of data to be sorted (transitions (2) and (3) in Figure 5).

4.2.4 Execution Plan for MapReduce

This section analyzes the execution plan generated by DryadLINQ for the MapReduce computation from Section 3.3. Here, we examine only the case when the input to `GroupBy` is not ordered and the reduce function is determined to be associative and commutative. The automatically generated execution plan is shown in Figure 6. The plan is statically transformed (1) into a Map and a Reduce stage. The Map stage applies the `SelectMany` operator (`SM`) and then sorts each partition (`S`), performs a local GroupBy (`G`) and finally a local reduction (`R`). The `D` nodes perform a hash-partition. All these operations are pipelined together in a single process. The Reduce stage first merge-sorts all the incoming data streams (`MS`). This is followed by another `GroupBy` (`G`) and the final reduction (`R`). All these Reduce stage operators are pipelined in a single process along with the subsequent operation in the computation (`X`). As with the sort plan

in Section 4.2.3, at run time (2) the number of Map instances is automatically determined using the number of input partitions, and the number of Reduce instances is chosen based on the total volume of data to be Reduced. If necessary, DryadLINQ will insert a dynamic aggregation tree (3) to reduce the amount of data that crosses the network. In the figure, for example, the two rightmost input partitions were processed on the same computer, and their outputs have been pre-aggregated locally before being transferred across the network and combined with the output of the leftmost partition.

The resulting execution plan is very similar to the manually constructed plans reported for Google's MapReduce [15] and the Dryad histogram computation in [26]. The crucial point to note is that in DryadLINQ MapReduce is not a primitive, hard-wired operation, and *all* user-specified computations gain the benefits of these powerful automatic optimization strategies.

4.3 Code Generation

The EPG is used to derive the Dryad execution plan after the static optimization phase. While the EPG encodes all the required information, it is not a runnable program. DryadLINQ uses dynamic code generation to automatically synthesize LINQ code to be run at the Dryad vertices. The generated code is compiled into a .NET assembly that is shipped to cluster computers at execution time. For each execution-plan stage, the assembly contains two pieces of code:

(1) The code for the LINQ subexpression executed by each node.

(2) Serialization code for the channel data. This code is much more efficient than the standard .NET serialization methods since it can rely on the contract between the reader and writer of a channel to access the same statically known datatype.

The subexpression in a vertex is built from pieces of the overall EPG passed in to DryadLINQ. The EPG is created in the original client computer's execution context, and may depend on this context in two ways:

(1) The expression may reference variables in the local context. These references are eliminated by partial evaluation of the subexpression at code-generation time. For primitive values, the references in the expressions are replaced with the actual values. Object values are serialized to a resource file which is shipped to computers in the cluster at execution time.

(2) The expression may reference .NET libraries. .NET reflection is used to find the transitive closure of all non-system libraries referenced by the executable, and these are shipped to the cluster computers at execution time.

4.4 Leveraging Other LINQ Providers

One of the greatest benefits of using the LINQ framework is that DryadLINQ can leverage other systems that use the same constructs. DryadLINQ currently gains most from the use of PLINQ [19], which allows us to run the code within each vertex in parallel on a multi-core server. PLINQ, like DryadLINQ, attempts to make the process of parallelizing a LINQ program as transparent as possible, though the systems' implementation strategies are quite different. Space does not permit a detailed explanation, but PLINQ employs the iterator model [25] since it is better suited to fine-grain concurrency in a shared-memory multi-processor system. Because both PLINQ and DryadLINQ use expressions composed from the same LINQ constructs, it is straightforward to combine their functionality. DryadLINQ's vertices execute LINQ expressions, and in general the addition by the DryadLINQ code generator of a single line to the vertex's program triggers the use of PLINQ, allowing the vertex to exploit all the cores in a cluster computer. We note that this remarkable fact stems directly from the careful design choices that underpin LINQ.

We have also added interoperation with the LINQ-to-SQL system which lets DryadLINQ vertices directly access data stored in SQL databases. Running a database on each cluster computer and storing tables partitioned across these databases may be much more efficient than using flat disk files for some applications. DryadLINQ programs can use "partitioned" SQL tables as input and output. DryadLINQ also identifies and ships some subexpressions to the SQL databases for more efficient execution.

Finally, the default single-computer LINQ-to-Objects implementation allows us to run DryadLINQ programs on a single computer for testing on small inputs under the control of the Visual Studio debugger before executing on a full cluster dataset.

4.5 Debugging

Debugging a distributed application is a notoriously difficult problem. Most DryadLINQ jobs are long running, processing massive datasets on large clusters, which could make the debugging process even more challenging. Perhaps surprisingly, we have not found debugging the correctness of programs to be a major challenge when using DryadLINQ. Several users have commented that LINQ's strong typing and narrow interface have turned up many bugs before a program is even executed. Also, as mentioned in Section 4.4, DryadLINQ supports a straightforward mechanism to run applications on a single computer, with very sophisticated support from the .NET development environment.

Once an application is running on the cluster, an individual vertex may fail due to unusual input data that manifests problems not apparent from a single-computer test. A consequence of Dryad's deterministic-replay execution model, however, is that it is straightforward to re-execute such a vertex in isolation with the inputs that caused the failure, and the system includes scripts to ship the vertex executable, along with the problematic partitions, to a local computer for analysis and debugging.

Performance debugging is a much more challenging problem in DryadLINQ today. Programs report summary information about their overall progress, but if particular stages of the computation run more slowly than expected, or their running time shows surprisingly high variance, it is necessary to investigate a collection of disparate logs to diagnose the issue manually. The centralized nature of the Dryad job manager makes it straightforward to collect profiling information to ease this task, and simplifying the analysis of these logs is an active area of our current research.

5 Experimental Evaluation

We have evaluated DryadLINQ on a set of applications drawn from domains including web-graph analysis, large-scale log mining, and machine learning. All of our performance results are reported for a medium-sized private cluster described in Section 5.1. Dryad has been in continuous operation for several years on production clusters made up of thousands of computers so we are confident in the scaling properties of the underlying execution engine, and we have run large-scale DryadLINQ programs on those production clusters.

5.1 Hardware Configuration

The experiments described in this paper were run on a cluster of 240 computers. Each of these computers was running the Windows Server 2003 64-bit operating system. The computers' principal components were two dual-core AMD Opteron 2218 HE CPUs with a clock speed of 2.6 GHz, 16 GBytes of DDR2 random access memory, and four 750 GByte SATA hard drives. The computers had two partitions on each disk. The first, small, partition was occupied by the operating system on one disk and left empty on the remaining disks. The remaining partitions on each drive were striped together to form a large data volume spanning all four disks. The computers were each connected to a Linksys SRW2048 48-port full-crossbar GBit Ethernet local switch via GBit Ethernet. There were between 29 and 31 computers connected to each local switch. Each local switch was in turn connected to a central Linksys SRW2048 switch, via 6 ports aggregated using 802.3ad link aggregation. This gave each local switch up to 6 GBits per second of full duplex connectivity. Note that the switches are commodity parts purchased for under $1000 each.

5.2 Terasort

In this experiment we evaluate DryadLINQ using the Terasort benchmark [3]. The task is to sort 10 billion 100-Byte records using case-insensitive string comparison on a 10-Byte key. We use the data generator described in [3]. The DryadLINQ program simply defines the record type, creates a `DryadTable` for the partitioned inputs, and calls `OrderBy`; the system then automatically generates an execution plan using dynamic range-partitioning as described in Section 4.2.3 (though for the purposes of running a controlled experiment we manually set the number of partitions for the sorting stage).

For this experiment, each computer in the cluster stored a partition of size around 3.87 GBytes ($4,166,666,600$ Bytes). We varied the number of computers used, so for an execution using n computers, the total data sorted is $3.87n$ GBytes. On the largest run $n = 240$ and we sort 10^{12} Bytes of data. The most time-consuming phase of this experiment is the network read to range-partition the data. However, this is overlapped with the sort, which processes inputs in batches in parallel and generates the output by merge-sorting the sorted batches. DryadLINQ automatically compresses the data before it is transferred across the network—when sorting 10^{12} Bytes of data, 150 GBytes of compressed data were transferred across the network.

Table 1 shows the elapsed times in seconds as the number of machines increases from 1 to 240, and thus the data sorted increases from 3.87 GBytes to 10^{12} Bytes. On repeated runs the times were consistent to within 5% of their averages. Figure 7 shows the same information in graphical form. For the case of a single partition, DryadLINQ uses a very different execution plan, skipping the sampling and partitioning stages. It thus reads the input data only once, and does not perform any network transfers. The single-partition time is therefore the baseline time for reading a partition, sorting it, and writing the output. For $2 \leq n \leq 20$ all computers were connected to the same local switch, and the elapsed time stays fairly constant. When $n > 20$ the elapsed time seems to be approaching an asymptote as we increase the number of computers. We interpret this to mean that the cluster is well-provisioned: we do not saturate the core

Computers	1	2	10	20	40	80	240
Time	119	241	242	245	271	294	319

Table 1: Time in seconds to sort different amounts of data. The total data sorted by an n-machine experiment is around $3.87n$ GBytes, or 10^{12} Bytes when $n = 240$.

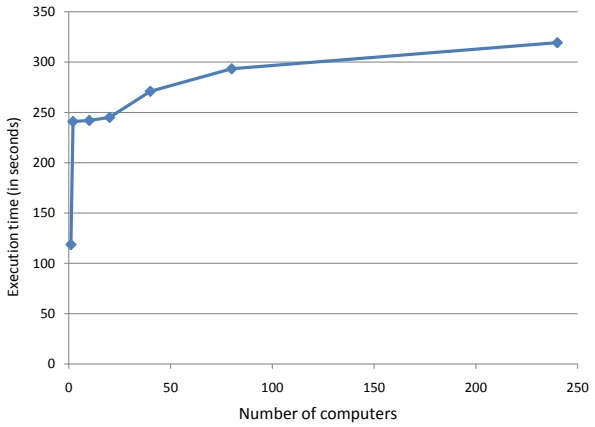


Figure 7: *Sorting increasing amounts of data while keeping the volume of data per computer fixed. The total data sorted by an n-machine experiment is around $3.87n$ GBytes, or 10^{12} Bytes when $n = 240$.*

Computers	1	5	10	20	40
Dryad	2167	451	242	135	92
DryadLINQ	2666	580	328	176	113

Table 2: Time in seconds to process skyserver Q18 using different number of computers.

network even when performing a dataset repartitioning across all computers in the cluster.

5.3 SkyServer

For this experiment we implemented the most time-consuming query (Q18) from the Sloan Digital Sky Survey database [23]. The query identifies a "gravitational lens" effect by comparing the locations and colors of stars in a large astronomical table, using a three-way Join over two input tables containing 11.8 GBytes and 41.8 GBytes of data, respectively. In this experiment, we compare the performance of the two-pass variant of the Dryad program described in [26] with that of DryadLINQ. The Dryad program is around 1000 lines of C++ code whereas the corresponding DryadLINQ program is only around 100 lines of C#. The input tables were manually range-partitioned into 40 partitions using the same keys. We varied n, the number of computers used, to investigate the scaling performance. For a given n we ensured that the tables were distributed such that each computer had approximately $40/n$ partitions of each, and that for a given partition key-range the data from the two tables was stored on the same computer.

Table 2 shows the elapsed times in seconds for the native Dryad and DryadLINQ programs as we varied n between 1 and 40. On repeated runs the times were consistent to within 3.5% of their averages. The DryadLINQ implementation is around 1.3 times slower than the native Dryad job. We believe the slowdown is mainly due

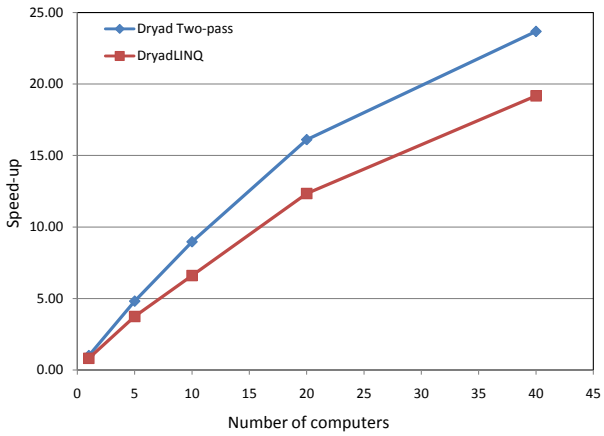


Figure 8: *The speed-up of the Skyserver Q18 computation as the number of computers is varied. The baseline is relative to DryadLINQ job running on a single computer and times are given in Table 2.*

to a hand-tuned sort strategy used by the Dryad program, which is somewhat faster than DryadLINQ's automatic parallel sort implementation. However, the DryadLINQ program is written at a much higher level. It abstracts much of the distributed nature of the computation from the programmer, and is only 10% of the length of the native code.

Figure 8 graphs the inverse of the running times, normalized to show the speed-up factor relative to the two-pass single-computer Dryad version. For $n \leq 20$ all computers were connected to the same local switch, and the speedup factor is approximately proportional to the number of computers used. When $n = 40$ the computers must communicate through the core switch and the scaling becomes sublinear.

5.4 PageRank

We also evaluate the performance of DryadLINQ at performing PageRank calculations on a large web graph. PageRank is a conceptually simple iterative computation for scoring hyperlinked pages. Each page starts with a real-valued score. At each iteration every page distributes its score across its outgoing links and updates its score to the sum of values received from pages linking to it. Each iteration of PageRank is a fairly simple relational query. We first Join the set of links with the set of ranks, using the source as the key. This results in a set of scores, one for each link, that we can accumulate using a GroupBy-Sum with the link's destinations as keys. We compare two implementations: an initial "naive" attempt and an optimized version.

Our first DryadLINQ implementation follows the outline above, except that the links are already grouped by source (this is how the crawler retrieves them). This makes the Join less complicated—once per page rather than once per link—but requires that we follow it with

a SelectMany, to produce the list of scores to aggregate. This naive implementation takes 93 lines of code, including 35 lines to define the input types.

The naive approach scales well, but is inefficient because it must reshuffle data proportional to the number of links to aggregate the transmitted scores. We improve on it by first HashPartitioning the link data by a hash of the hostname of the source, rather than a hash of the page name. The result is that most of the rank updates are written back locally—80%-90% of web links are host-local—and do not touch the network. It is also possible to cull leaf pages from the graph (and links to them); they do not contribute to the iterative computation, and needn't be kept in the inner loop. Further performance optimizations, like pre-grouping the web pages by host (+7 LOC), rewriting each of these host groups with dense local names (+21 LOC), and pre-aggregating the ranks from each host (+18 LOC) simplify the computation further and ease the memory footprint. The complete source code for our implementation of PageRank is contained in the companion technical report [38].

We evaluate both of these implementations (running on 240 computers) on a large web graph containing 954M pages and 16.5B links, occupying 1.2 TB compressed on disk. The naive implementation, including pre-aggregation, executes 10 iterations in 12,792 seconds. The optimized version, which further compresses the graph down to 116 GBytes, executes 10 iterations in 690 seconds.

It is natural to compare our PageRank implementation with similar implementations using other platforms. MapReduce, Hadoop, and Pig all use the MapReduce computational framework, which has trouble efficiently implementing Join due to its requirement that all input (including the web graph itself) be output of the previous stage. By comparison, DryadLINQ can partition the web graph once, and reuse that graph in multiple stages without moving any data across the network. It is important to note that the Pig language masks the complexity of Joins, but they are still executed as MapReduce computations, thus incurring the cost of additional data movement. SQL-style queries can permit Joins, but suffer from their rigid data types, preventing the pre-grouping of links by host and even by page.

5.5 Large-Scale Machine Learning

We ran two machine-learning experiments to investigate DryadLINQ's performance on iterative numerical algorithms.

The first experiment is a clustering algorithm for detecting botnets. We analyze around 2.1 GBytes of data, where each datum is a three-dimensional vector summarizing salient features of a single computer, and group them into 3 clusters. The algorithm was written using the machine-learning framework described in Section 3.3 in 160 lines of C#. The computation has three stages: (1) parsing and re-partitioning the data across all the computers in the cluster; (2) counting the records; and (3) performing an iterative E–M computation. We always perform 10 iterations (ignoring the convergence criterion) grouped into two blocks of 5 iterations, materializing the results every 5 iterations. Some stages are CPU-bound (performing matrix algebra), while other are I/O bound. The job spawns about 10,000 processes across the 240 computers, and completes end-to-end in 7 minutes and 11 seconds, using about 5 hours of effective CPU time.

We also used DryadLINQ to apply statistical inference algorithms [33] to automatically discover network-wide relationships between hosts and services on a medium-size network (514 hosts). For each network host the algorithms compose a dependency graph by analyzing timings between input/output packets. The input is processed header data from a trace of 11 billion packets (180 GBytes packed using a custom compression format into 40 GBytes). The main body of this DryadLINQ program is just seven lines of code. It hash-partitions the data using the pair `(host,hour)` as a key, applies a doubly-nested E–M algorithm and hypothesis testing (which takes 95% of the running time), partitions again by hour, and finally builds graphs for all 174,588 active host hours. The computation takes 4 hours and 22 minutes, and more than 10 days of effective CPU time.

6 Related Work

DryadLINQ builds on decades of previous work in distributed computation. The most obvious connections are with parallel databases, grid and cluster computing, parallel and high-performance computation, and declarative programming languages.

Many of the salient features of DryadLINQ stem from the high-level system architecture. In our model of cluster computing the three layers of storage, execution, and application are decoupled. The system can make use of a variety of storage layers, from raw disk files to distributed filesystems and even structured databases. The Dryad distributed execution environment provides generic distributed execution services for acyclic networks of processes. DryadLINQ supplies the application layer.

6.1 Parallel Databases

Many of the core ideas employed by DryadLINQ (such as shared-nothing architecture, horizontal data partitioning, dynamic repartitioning, parallel query evaluation,

and dataflow scheduling), can be traced to early research projects in parallel databases [18], such as Gamma [17], Bubba [8], and Volcano [22], and found in commercial products for data warehousing such as Teradata, IBM DB2 Parallel Edition [4], and Tandem SQL/MP [20].

Although DryadLINQ builds on many of these ideas, it is not a traditional database. For example, DryadLINQ provides a generalization of the concept of query language, but it does not provide a data definition language (DDL) or a data management language (DML) and it does not provide support for in-place table updates or transaction processing. We argue that the DDL and DML belong to the storage layer, so they should not be a first-class part of the application layer. However, as Section 4.2 explains, the DryadLINQ optimizer does make use of partitioning and typing information available as metadata attached to input datasets, and will write such metadata back to an appropriately configured storage layer.

Traditional databases offer extensibility beyond the simple relational data model through embedded languages and stored procedures. DryadLINQ (following LINQ's design) turns this relationship around, and embeds the expression language in the high-level programming language. This allows DryadLINQ to provide very rich native datatype support: almost all native .NET types can be manipulated as typed, first-class objects.

In order to enable parallel expression execution, DryadLINQ employs many traditional parallelization and query optimization techniques, centered on horizontal data partitioning. As mentioned in the Introduction, the expression plan generated by DryadLINQ is virtualized. This virtualization underlies DryadLINQ's dynamic optimization techniques, which have not previously been reported in the literature [16].

6.2 Large Scale Data-Parallel Computation Infrastructure

The last decade has seen a flurry of activity in architectures for processing very large datasets (as opposed to traditional high-performance computing which is typically CPU-bound). One of the earliest commercial generic platforms for distributed computation was the Teoma Neptune platform [13], which introduced a map-reduce computation paradigm inspired by MPI's Reduce operator. The Google MapReduce framework [15] slightly extended the computation model, separated the execution layer from storage, and virtualized the execution. The Hadoop open-source port of MapReduce uses the same architecture. NetSolve [5] proposed a grid-based architecture for a generic execution layer. DryadLINQ has a richer set of operators and better language support than any of these other proposals.

At the storage layer a variety of very large scale simple databases have appeared, including Google's BigTable [11], Amazon's Simple DB, and Microsoft SQL Server Data Services. Architecturally, DryadLINQ is just an application running on top of Dryad and generating distributed Dryad jobs. We can envision making it interoperate with any of these storage layers.

6.3 Declarative Programming Languages

Notable research projects in parallel declarative languages include Parallel Haskell [37], Cilk [7], and NESL [6].

There has also been a recent surge of activity on layering distributed and declarative programming language on top of distributed computation platforms. For example, Sawzall [32] is compiled to MapReduce applications, while Pig [31] programs are compiled to the Hadoop infrastructure. The MapReduce model is extended to support Joins in [12]. Other examples include Pipelets [9], HIVE (an internal Facebook language built on Hadoop), and Scope [10], Nebula [26], and PSQL (internal Microsoft languages built on Dryad).

Grid computing usually provides workflows (and not a programming language interface), which can be tied together by a user-level application. Examples include Swift [39] and its scripting language, Taverna [30], and Triana [36]. DryadLINQ is a higher-level language, which better conceals the underlying execution fabric.

7 Discussion and Conclusions

DryadLINQ has been in use by a small community of developers for over a year, resulting in tens of large applications and many more small programs. The system was recently released more widely within Microsoft and our experience with it is rapidly growing as a result. Feedback from users has generally been very positive. It is perhaps of particular interest that most of our users manage small private clusters of, at most, tens of computers, and still find substantial benefits from DryadLINQ.

Of course DryadLINQ is not appropriate for all distributed applications, and this lack of generality arises from design choices in both Dryad and LINQ.

The Dryad execution engine was engineered for batch applications on large datasets. There is an overhead of at least a few seconds when executing a DryadLINQ EPG which means that DryadLINQ would not currently be well suited to, for example, low-latency distributed database lookups. While one could imagine re-engineering Dryad to mitigate some of this latency, an effective solution would probably need to adopt different strategies for, at least, resource virtualization, fault-

tolerance, and code generation and so would look quite different to our current system.

The question of which applications are suitable for parallelization by DryadLINQ is more subtle. In our experience, the main requirement is that the program can be written using LINQ constructs: users generally then find it straightforward to adapt it to distributed execution using DryadLINQ—and in fact frequently no adaptation is necessary. However, a certain change in outlook may be required to identify the data-parallel components of an algorithm and express them using LINQ operators. For example, the PageRank computation described in Section 5 uses a Join operation to implement a subroutine typically specified as matrix multiplication.

Dryad and DryadLINQ are also inherently specialized for streaming computations, and thus may appear very inefficient for algorithms which are naturally expressed using random-accesses. In fact for several workloads including breadth-first traversal of large graphs we have found DryadLINQ outperforms specialized random-access infrastructures. This is because the current performance characteristics of hard disk drives ensures that sequential streaming is faster than small random-access reads even when greater than 99% of the streamed data is discarded. Of course there will be other workloads where DryadLINQ is much less efficient, and as more storage moves from spinning disks to solid-state (e.g. flash memory) the advantages of streaming-only systems such as Dryad and MapReduce will diminish.

We have learned a lot from our users' experience of the `Apply` operator. Many DryadLINQ beginners find it easier to write custom code inside `Apply` than to determine the equivalent native LINQ expression. `Apply` is therefore helpful since it lowers the barrier to entry to use the system. However, the use of `Apply` "pollutes" the relational nature of LINQ and can reduce the system's ability to make high-level program transformations. This tradeoff between purity and ease of use is familiar in language design. As system builders we have found one of the most useful properties of `Apply` is that sophisticated programmers can use it to manually implement optimizations that DryadLINQ does not perform automatically. For example, the optimizer currently implements all reductions using partial sorts and groupings as shown in Figure 6. In some cases operations such as Count are much more efficiently implemented using hash tables and accumulators, and several developers have independently used `Apply` to achieve this performance improvement. Consequently we plan to add additional reduction patterns to the set of automatic DryadLINQ optimizations. This experience strengthens our belief that, at the current stage in the evolution of the system, it is best to give users flexibility and suffer the consequences when they use that flexibility unwisely.

DryadLINQ has benefited tremendously from the design choices of LINQ and Dryad. LINQ's extensibility, allowing the introduction of new execution implementations and custom operators, is the key that allows us to achieve deep integration of Dryad with LINQ-enabled programming languages. LINQ's strong static typing is extremely valuable when programming large-scale computations—it is much easier to debug compilation errors in Visual Studio than run-time errors in the cluster. Likewise, Dryad's flexible execution model is well suited to the static and dynamic optimizations we want to implement. We have not had to modify any part of Dryad to support DryadLINQ's development. In contrast, many of our optimizations would have been difficult to express using a more limited computational abstraction such as MapReduce.

Our current research focus is on gaining more understanding of what programs are easy or hard to write with DryadLINQ, and refining the optimizer to ensure it deals well with common cases. As discussed in Section 4.5, performance debugging is currently not well supported. We are working to improve the profiling and analysis tools that we supply to programmers, but we are ultimately more interested in improving the system's ability to get good performance automatically. We are also pursuing a variety of cluster-computing projects that are enabled by DryadLINQ, including storage research tailored to the workloads generated by DryadLINQ applications.

Our overall experience is that DryadLINQ, by combining the benefits of LINQ—a high-level language and rich data structures—with the power of Dryad's distributed execution model, proves to be an amazingly simple, useful and elegant programming environment.

Acknowledgements

We would like to thank our user community for their invaluable feedback, and particularly Frank McSherry for the implementation and evaluation of PageRank. We also thank Butler Lampson, Dave DeWitt, Chandu Thekkath, Andrew Birrell, Mike Schroeder, Moises Goldszmidt, and Kannan Achan, as well as the OSDI review committee and our shepherd Marvin Theimer, for many helpful comments and contributions to this paper.

References

[1] The DryadLINQ project. http://research.microsoft.com/research/sv/DryadLINQ/.

[2] The LINQ project. http://msdn.microsoft.com/netframework/future/linq/.

[3] Sort benchmark. http://research.microsoft.com/barc/SortBenchmark/.

[4] BARU, C. K., FECTEAU, G., GOYAL, A., HSIAO, H., JHINGRAN, A., PADMANABHAN, S., COPELAND, G. P., AND WIL-

son, W. G. DB2 parallel edition. *IBM Systems Journal 34*, 2, 1995.

[5] Beck, M., Dongarra, J., and Plank, J. S. NetSolve/D: A massively parallel grid execution system for scalable data intensive collaboration. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[6] Blelloch, G. E. Programming parallel algorithms. *Communications of the ACM (CACM) 39*, 3, 1996.

[7] Blumofe, R. D., Joerg, C. F., Kuszmaul, B., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: An efficient multithreaded runtime system. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1995.

[8] Boral, H., Alexander, W., Clay, L., Copeland, G., Danforth, S., Franklin, M., Hart, B., Smith, M., and Valduriez, P. Prototyping Bubba, a highly parallel database system. *IEEE Trans. on Knowl. and Data Eng. 2*, 1, 1990.

[9] Carnahan, J., and DeCoste, D. Pipelets: A framework for distributed computation. In *W4: Learning in Web Search*, 2005.

[10] Chaiken, R., Jenkins, B., Larson, P.-Å., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J. SCOPE: Easy and efficient parallel processing of massive data sets. In *International Conference of Very Large Data Bases (VLDB)*, 2008.

[11] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. BigTable: A distributed storage system for structured data. In *Symposium on Operating System Design and Implementation (OSDI)*, 2006.

[12] chih Yang, H., Dasdan, A., Hsiao, R.-L., and Parker, D. S. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD international conference on Management of data*, 2007.

[13] Chu, L., Tang, H., Yang, T., and Shen, K. Optimizing data aggregation for cluster-based internet services. In *Symposium on Principles and practice of parallel programming (PPoPP)*, 2003.

[14] Cruanes, T., Dageville, B., and Ghosh, B. Parallel SQL execution in Oracle 10g. In *ACM SIGMOD*, 2004.

[15] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[16] Deshpande, A., Ives, Z., and Raman, V. Adaptive query processing. *Foundations and Trends in Databases 1*, 1, 2007.

[17] DeWitt, D., Ghandeharizadeh, S., Schneider, D., Hsiao, H., Bricker, A., and Rasmussen, R. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering 2*, 1, 1990.

[18] DeWitt, D., and Gray, J. Parallel database systems: The future of high performance database processing. *Communications of the ACM 36*, 6, 1992.

[19] Duffy, J. A query language for data parallel programming. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, 2007.

[20] Englert, S., Glasstone, R., and Hasan, W. Parallelism and its price : A case study of nonstop SQL/MP. In *Sigmod Record*, 1995.

[21] Feng, L., Lu, H., Tay, Y. C., and Tung, A. K. H. Buffer management in distributed database systems: A data mining based approach. In *International Conference on Extending Database Technology*, 1998, H.-J. Schek, F. Saltor, I. Ramos, and G. Alonso, Eds., vol. 1377 of *Lecture Notes in Computer Science*.

[22] Graefe, G. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD International Conference on Management of data*, 1990.

[23] Gray, J., Szalay, A., Thakar, A., Kunszt, P., Stoughton, C., Slutz, D., and Vandenberg, J. Data mining the SDSS SkyServer database. In *Distributed Data and Structures 4: Records of the 4th International Meeting*, 2002.

[24] Hasan, W., Florescu, D., and Valduriez, P. Open issues in parallel query optimization. *SIGMOD Rec. 25*, 3, 1996.

[25] Hellerstein, J. M., Stonebraker, M., and Hamilton, J. Architecture of a database system. *Foundations and Trends in Databases 1*, 2, 2007.

[26] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of European Conference on Computer Systems (EuroSys)*, 2007.

[27] Kabra, N., and DeWitt, D. J. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD International Conference on Management of Data*, 1998.

[28] Kossmann, D. The state of the art in distributed query processing. *ACM Comput. Surv. 32*, 4, 2000.

[29] Morvan, F., and Hameurlain, A. Dynamic memory allocation strategies for parallel query execution. In *Symposium on Applied computing (SAC)*, 2002.

[30] Oinn, T., Greenwood, M., Addis, M., Ferris, J., Glover, K., Goble, C., Hull, D., Marvin, D., Li, P., Lord, P., Pocock, M. R., Senger, M., Wipat, A., and Wroe, C. Taverna: Lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience 18*, 10, 2005.

[31] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. Pig Latin: A not-so-foreign language for data processing. In *International Conference on Management of Data (Industrial Track) (SIGMOD)*, 2008.

[32] Pike, R., Dorward, S., Griesemer, R., and Quinlan, S. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming 13*, 4, 2005.

[33] Simma, A., Goldszmidt, M., MacCormick, J., Barham, P., Black, R., Isaacs, R., and Mortier, R. CT-NOR: representing and reasoning about events in continuous time. In *International Conference on Uncertainty in Artificial Intelligence*, 2008.

[34] Stonebraker, M., Bear, C., Çetintemel, U., Cherniack, M., Ge, T., Hachem, N., Harizopoulos, S., Lifter, J., Rogers, J., and Zdonik, S. One size fits all? Part 2: Benchmarking results. In *Conference on Innovative Data Systems Research (CIDR)*, 2005.

[35] 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 *International Conference of Very Large Data Bases (VLDB)*, 2007.

[36] Taylor, I., Shields, M., Wang, I., and Harrison, A. *Workflows for e-Science*. 2007, ch. The Triana Workflow Environment: Architecture and Applications, pp. 320–339.

[37] Trinder, P., Loidl, H.-W., and Pointon, R. Parallel and distributed Haskells. *Journal of Functional Programming 12*, (4&5), 2002.

[38] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P. K., Currey, J., McSherry, F., and Achan, K. Some sample programs written in DryadLINQ. Tech. Rep. MSR-TR-2008-74, Microsoft Research, 2008.

[39] Zhao, Y., Hategan, M., Clifford, B., Foster, I., von Laszewski, G., Nefedova, V., Raicu, I., Stef-Praun, T., and Wilde, M. Swift: Fast, reliable, loosely coupled parallel computation. *IEEE Congress on Services*, 2007.

Everest: Scaling down peak loads through I/O off-loading

Dushyanth Narayanan Austin Donnelly Eno Thereska Sameh Elnikety
Antony Rowstron
Microsoft Research Cambridge, United Kingdom
{*dnarayan,austind,etheres,samehe,antr*}*@microsoft.com*

Abstract

Bursts in data center workloads are a real problem for storage subsystems. Data volumes can experience peak I/O request rates that are over an order of magnitude higher than average load. This requires significant overprovisioning, and often still results in significant I/O request latency during peaks.

In order to address this problem we propose Everest, which allows data written to an overloaded volume to be temporarily off-loaded into a short-term virtual store. Everest creates the short-term store by opportunistically pooling underutilized storage resources either on a server or across servers within the data center. Writes are temporarily off-loaded from overloaded volumes to lightly loaded volumes, thereby reducing the I/O load on the former. Everest is transparent to and usable by unmodified applications, and does not change the persistence or consistency of the storage system. We evaluate Everest using traces from a production Exchange mail server as well as other benchmarks: our results show a 1.4–70 times reduction in mean response times during peaks.

1 Introduction

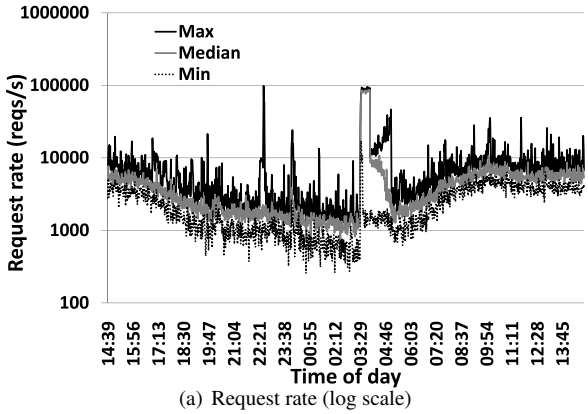
Many server I/O workloads are bursty, characterized as having peak I/O loads significantly higher than the average load. If the storage subsystem is not provisioned for its peak load, its performance during peaks degrades significantly, resulting in I/O operations having significant latency. We observe that workloads are usually unbalanced across servers in a data center, and often even across the data volumes associated with a single server. We propose Everest, a system that improves the performance of overloaded volumes by transparently exploiting statistical multiplexing across the storage bandwidth resources in the data center.

Everest monitors the performance of a data volume, and if the load on the volume increases beyond a predefined threshold, it utilizes spare bandwidth on other storage volumes to absorb writes performed to the overloaded volume. It does this by maintaining a *virtual short-term persistent store*, into which data is temporarily written, or off-loaded. The store is virtual in the sense that storage resources are not explicitly allocated to it; rather it is created by pooling idle bandwidth and spare capacity on existing data volumes either on a single server or across a set of servers in the same data center. In the common case, this can remove the majority of writes from the peak load, allowing the data volume under stress to serve mostly reads. When the peak subsides, the off-loaded data is lazily reclaimed back to the original volume, freeing the space in the Everest store. Everest handles short-term peaks and is not designed to handle long-term changes in load: these must be addressed by reprovisioning the storage subsystem and changing the data layout to match the new workload patterns.

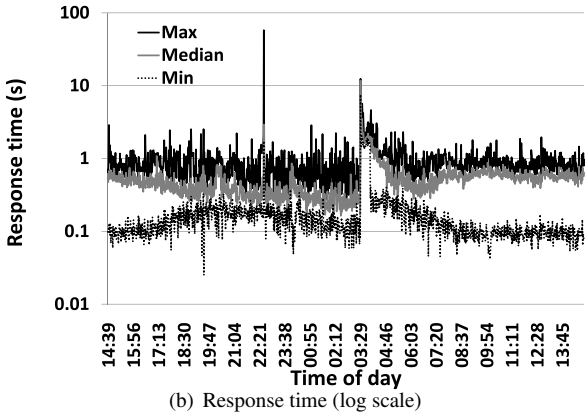
Everest thus provides a short-term, low-latency persistent store without the requirements for explicitly provisioned resources. Everest is interposed at the block I/O interface level, and is transparent to the applications and services running above it. It does not alter the persistence or consistency semantics of the storage subsystem, and unmodified applications can use Everest.

Two recent developments make storage pooling for peak I/O loads increasingly important. First, gigabit networking is ubiquitous in today's data centers, with servers configured with multiple high bandwidth NICs, and soon 10-gigabit networks will be common. This increase in network bandwidth relative to storage bandwidth is exploited by many storage technologies, such as Network Attached Storage (NAS) and Storage Area Networks (SAN). Everest also exploits it by allowing I/O off-loading across the network.

Second, as disk bandwidth increases more slowly than capacity, bandwidth rather than capacity increasingly determines the number of disks provisioned for an application. Peak off-loading avoids the need to provision each data volume individually for its peak load, which is often



(a) Request rate (log scale)



(b) Response time (log scale)

Figure 1: Exchange request rate and response time over a 24-hour period, 12–13 December 2007.

impossible (since peaks are unpredictable) and always expensive. Instead the volumes can be configured for the expected mean or 95th percentile I/O rate, and peak off-loading used to improve performance during periods when the I/O rate is higher than expected.

We evaluate the increase in performance Everest achieves using real world traces gathered from a production Exchange mail server, as well as database benchmarks and micro-benchmarks. The results show that Everest provides significant benefit. For example, for the mail server the mean response time during the worst peak is reduced by a factor of 70. In tests of database OLTP throughput, off-loading from a loaded machine to one idle machine increased throughput by a factor of 3, scaling up to a factor of 6 with 3 idle machines.

The rest of the paper is organized as follows. Section 2 provides further background, then Section 3 describes the design and implementation of Everest. Section 4 presents evaluation results, Section 5 related work, and Section 6 concludes the paper.

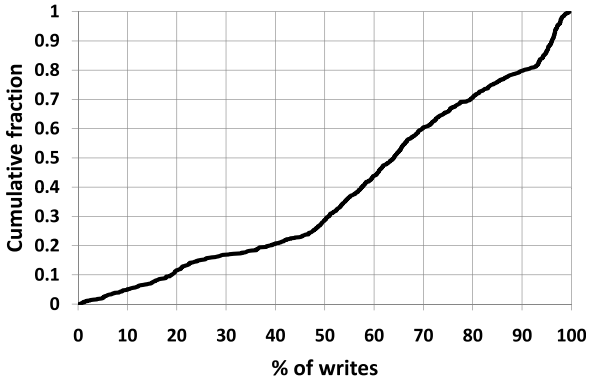


Figure 2: % of writes in Exchange: CDF over 1 minute intervals having mean response times above 1 second.

2 Background

In order to understand the impact of peak I/O loads, we examined a trace of a production Exchange e-mail server running inside Microsoft [26]. Employee e-mail at Microsoft is supported by a number of such servers, which are provisioned and maintained by the corporate IT department. The trace records the I/Os issued by one such server, which supports 5000 users. The trace covers 8 data volumes with a total capacity of 7.2 TB, for a 24-hour period starting at 14:39 on the 12th December 2007.

For each minute in the trace, we measured the mean request rate for each of the volumes, and Figure 1(a) shows for each minute the maximum and minimum request rates across all volumes, as well as the request rate for the median volume. It should be noted that due to the large variations in rates a log scale is used. The load is extremely bursty, and also unevenly distributed across volumes during peak bursts. Across the entire trace the peak-to-mean ratio in the I/O load is 13.5, and during the highest peak 90% of the load is on a single volume.

To understand the impact this load has on response times, we calculated the mean response time over 1 minute intervals for each data volume. Figure 1(b) shows the maximum, median, and minimum response times across volumes, again using a log scale. As expected, the response times vary widely, from under a second in the common case to above 10 seconds during the peaks. Further, there is substantial variation across volumes when at peak. This implies that, even on a single server, there is scope for statistical multiplexing of disk bandwidth during peak load episodes.

For Everest to be able to provide benefit, the I/O peaks must contain writes as well as reads. We believe peaks in server workloads are likely to have a significant fraction of writes. Storage subsystems are typically well-equipped to handle large streaming read workloads, and small random-access reads will benefit from caching at

various levels of the system. Figure 2 shows the fraction of write requests for every 1 minute interval in the Exchange traces in which the mean response time exceeded 1 second, as a cumulative distribution. Fewer than 5% of these intervals have less than 10% writes.

Battery-backed non-volatile RAM (NVRAM) is sometimes used in storage systems to improve I/O performance, although its use is often limited due to the cost and the need for maintenance of the batteries. NVRAM can only provide benefit when the I/O peak's footprint is smaller than the NVRAM size. The Exchange mail server that we traced is configured with 512 MB of NVRAM shared across all volumes. Figure 1 clearly shows that the response times are high despite the use of NVRAM. In general, it is not cost-effective to provision sufficient NVRAM to handle worst-case peak loads.

3 Design

Off-loading in Everest is configured on a per-volume basis. Here by volume we mean any block storage device: e.g., a single disk, array of disks, or solid-state storage device (SSD). An Everest *client* can be associated with any volume, which we then call the *base volume* for that client. The client is interposed on all read and write requests to the base volume. When the base volume is overloaded, the client off-loads writes into a virtual store. This allows more of the base volume's bandwidth to be used for servicing reads. When the load peak subsides, the client *reclaims* the data in the background from the virtual store to the base volume.

The virtual store is formed by pooling many individual physical stores, referred to simply as *stores* in this paper. Each store has an underlying base volume and uses a small partition or file on this volume. Thus stores are not explicitly provisioned with resources but export spare capacity and idle bandwidth on existing volumes. When a store's base volume is idle, it is used opportunistically by clients whose base volume is heavily loaded.

Each Everest client is configured to use some set of stores, called its *store set*. The stores can be on different volumes on the same server as the client, or on different servers in the data center. A single volume can host a client, a store, or both. In general a client can use any store, and a store can be used by multiple clients. However, a client would not be configured to use a store having the same base volume as itself, since this does not contribute any additional disk bandwidth.

Figure 3 shows an example of a server in a data center using Everest. The server has two volumes, each of which is an array of many disks. Both volumes are configured with Everest clients, which interpose between the volumes and the file system and appear as standard block devices to the file system. Client 1 is configured to use

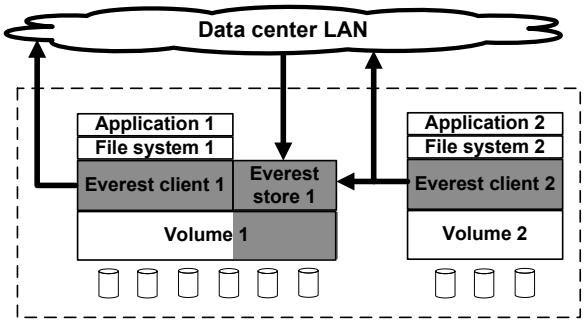


Figure 3: Example server running Everest.

Everest stores on other servers in the data center; client 2 is configured to use the store on volume 1 as well as stores on other servers.

The key challenges for Everest are to retain the consistency and persistence guarantees of the base volume while providing low latency and high throughput for off-loaded writes. This is challenging because Everest needs to maintain a consistent view of the data logically stored on each data volume, even though it may be physically distributed across several other volumes and servers when off-loaded. While data is being off-loaded there can be no synchronous writes of meta-data to the original volume, as this would increase its I/O load when already overloaded. Further, different versions of the same data could be off-loaded to multiple different locations during the same I/O peak, and a read request might span data that is off-loaded to different locations. The system must always return the latest version of the data in all cases. The persistent state of the system must also be recoverable after a failure.

Everest meets these goals through the combination of techniques described in the rest of this section. Much of the complexity in the Everest design is in the client, which is responsible for deciding when to off-load, where to off-load, and when to reclaim off-loaded data. The client is also responsible for consistency and fault-tolerance. We first describe the simpler Everest store and then the more complex client.

3.1 Everest store

An Everest store provides short-term write-optimized storage. It exports four operations: $write$, $read$, $read\_any$, and $delete$. Clients call $write$ when off-loading write requests. $read$ is invoked on a store when a client receives a read request for data off-loaded to that store. This happens rarely since blocks are only off-loaded for short periods, and are likely to remain in application and file system buffer caches during these periods. $read\_any$ and $delete$ are background operations used by clients to reclaim off-loaded data from the store.

The Everest store is optimized for low latency and high throughput of the frequently called foreground operation: $write$. The store uses a circular log layout to achieve sequential performance on writes. Each $write$ request results in a single write to the log head of a record containing both data and meta-data. The meta-data is also cached in memory for fast lookup. Background delete requests also cause records to be written to the log head; deletion records contain only meta-data and no data. If there are multiple concurrent $write$ requests, the store issues them concurrently to the disk to maximize performance. Write acknowledgements are serialized in log sequence order: a write is acknowledged only after all writes preceding it in the log are persistent on disk. This ensures that there are no holes in the log, and that all acknowledged writes are recoverable after a crash.

In the common case, the store absorbs a write burst, causing the head of the log to move forward. Subsequently clients reclaim and delete the data; this moves the tail forward and shrinks the log which eventually becomes empty. Thus both the head and the tail of the log move forward, wrapping around to the beginning when they reach the end of the file or partition. The head of the log is never allowed to wrap past the tail. In the common case the log does not fill up the allocated storage capacity. If the store does reach its capacity limit, it stops accepting write requests but continues to accept read and delete requests, which will eventually shrink the log.

The Everest store writes meta-data in each log record which allows it to correctly recover its state after a crash. The meta-data contains the ID of the client that issued the write, the block range written, the version, and a data checksum. State is recovered after a crash by scanning the log from tail to head. A pointer to the log tail is stored at the beginning of the store's file or partition. A write or delete record written to the log head might cause older records in the log to become stale; when the tail record becomes stale, the tail pointer can be moved forward. Tail pointer updates are done lazily during idle periods to avoid contention with other requests on the volume.

The head of the log is the last valid record read during the recovery scan. Partially written log records are detected by verifying the checksum in the record header. Over time, the log head might cycle repeatedly around the storage area. Hence, in general, the disk blocks following the log head could contain arbitrary data. To distinguish arbitrary data from valid log records, the store uses a 128-bit *epoch ID* that is randomly generated every time the log head wraps around. Each record that is appended to the log contains both its own epoch ID as well as that of the previous record. This property is verified during recovery, ensuring that only valid records are used to reconstruct the state after a failure.

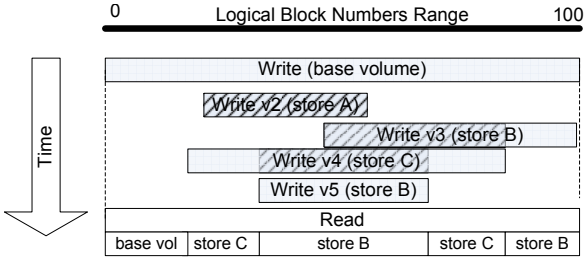


Figure 4: Example of range map holding versions: hashed areas represent stale/overwritten subranges.

3.2 Everest client

The Everest client is responsible for

1. off-loading writes to stores in its store set,
2. reclaiming data from the stores, and
3. guaranteeing persistence and consistency.

Achieving these goals is challenging for two reasons. First, when off-loading writes, the client must maintain consistency without writing either data or meta-data to the already overloaded base volume. Second, off-loaded data could be spread over multiple Everest stores; the client must correctly redirect read requests to the appropriate combination of base volume and/or Everest stores.

Since Everest interposes transparently above a block device, it provides the same consistency and persistence semantics as a local block device. The data returned for any read request is always the result of the last acknowledged write to that block, or of some issued but unacknowledged write. This property holds across transient failures, i.e., after a crash or reboot. Everest does *not* implement sharing of data across clients: this must be done in a higher layer if desired. At the Everest level a client mediates all access to its base volume and hence owns the namespace of blocks on the volume.

This section describes the key features of the Everest client: recoverable soft state, load balancing, reclaiming, and N-way off-loading for fault-tolerance.

3.2.1 Recoverable soft state

To avoid writing meta-data to the base volume when it is loaded, the client keeps almost all its meta-data as in-memory soft state. The only persistent state kept on the base volume is the store set: the list of Everest stores that the client can off-load to. This set changes infrequently, and is not changed during load peaks.

The soft state for each client contains an entry for each range of blocks off-loaded to an Everest store, which specifies the range, the store holding the latest version, and the version. The soft state is accessed on each I/O intercepted by the client, and is designed to be compact as well as efficient to query and update. It is maintained as

a *range map*, which contains ordered, non-overlapping ranges of byte offsets in a logarithmic search tree [20]. This supports fast lookup and update while using memory linear in the number of distinct block ranges off-loaded, rather than in the number of blocks. The range map can be queried to find the ranges that are currently off-loaded, and the Everest store holding the latest version. Range queries and updates can overlap arbitrarily, for example an update could partially overwrite an existing range with a newer version: the range map handles this by splitting ranges appropriately. Figure 4 shows an example of the in-memory state of a client after multiple overlapping writes have been completed.

On each read request, the client queries the range map to find out how to split the read between stores holding off-loaded blocks (if any), and the base volume. On each write request, the client queries the range map to see if the write request overlaps any currently off-loaded blocks. When an off-loaded write is completed, the client updates the range map before acknowledging the write completion to the higher layers.

An Everest client can correctly recover its soft state after a crash. Each off-loaded write sent to an Everest store is written along with meta-data that identifies the client, the base volume, the block range written, and the version. Keeping meta-data on the Everest stores allows the client to quickly and efficiently recover the soft state. After a failure the client retrieves meta-data in parallel from all the stores in its store set. This meta-data is cached by each store in a per-client range map. The responses from the stores are then merged to construct the client's state. If a store crashes and recovers concurrently with the client, the client waits for the store to recover by scanning its log, and then retrieves its meta-data.

3.2.2 Load balancing

In general, when an Everest client receives a write request, it can choose to send the write to the base volume or to any store in its store set. In making this choice, the client has four goals. First, it must always maintain correctness. Second, it should only off-load when the base volume is overloaded: unnecessary off-loading wastes resources and increases the chances of contention between workloads. Third, it should only off-load to stores on lightly loaded volumes, to avoid degrading the performance of workloads using those volumes. Finally, subject to these constraints, it should balance load between the available stores and the base volume.

Correctness requires that reads always go to the location holding the latest version of the data: the client splits read requests as necessary to achieve this. Most reads will go to the base volume, since reads of recently off-loaded data are rare. Most writes can be sent either to the base volume or to any available store in the store set. However a write that overwrites currently off-loaded data must also be off-loaded to some store, not written to the base volume. This is because the stores support versioning, but the base volume is a standard block device and hence cannot be assumed to support versioning. Hence if any version of the data is currently off-loaded, then the latest version must also be off-loaded to ensure that the client's state is correctly recoverable.

While the client generally has no choice on where to send read requests, it can redirect most write requests according to load. If a write overlaps a currently off-loaded range, then it is sent to the least loaded available store. Otherwise, if the load on the base volume is above a threshold T_{base}, and the least-loaded store has a load lower than another threshold T_{store}, writes are sent to the least loaded of the base volume and the available stores. Otherwise, writes are sent to the base volume.

Each Everest store periodically broadcasts load updates on the network and updates are also piggybacked on response packets sent to clients. The updates contain several block device level load metrics such as request rates, response times, and queue lengths. Thus a variety of load balancing metrics and policies are possible. Currently we use the simple policy described above, and we use the queue length — the number of I/O requests in flight — as the load metric.

3.2.3 Reclaiming

When the base volume load is below the threshold T_{base}, the Everest client *reclaims* off-loaded data to the base volume in the background. The client issues low-priority $read\_any$ requests to all stores holding valid data for it. Each lightly-loaded store returns valid data and meta-data (if any) for some range of blocks off-loaded by the client to that store. In general a store's log might contain many records corresponding to a given client; the store chooses the record that is closest to the log tail and contains some valid (non-stale) data for the client. The client writes the data to the base volume, and then sends a deletion request to the store. The client sends such deletion requests to a store whenever data on it becomes stale, either due to a reclaim or due to a newer version being written to a different store.

In some rare cases the reclaim process results in wasted I/O; however consistency is always maintained and all off-loaded data is eventually reclaimed. For example, while a block is being reclaimed, an application might issue a new write for the same block: any work done to reclaim the old version will be wasted. Correctness is maintained through three invariants. First, the client will send a deletion request for a version v only if the corresponding data has been written to the base

volume, or a version $v' > v$ has been written to some store. Second, if a version v of a block is currently off-loaded, then the client will off-load any new write to that block, with a higher version $v' > v$. Third, the client will not send a deletion request for the highest currently off-loaded version of any block until all older versions of the block have been deleted from all stores.

The Everest client performs multiple concurrent reclaim operations for efficient pipelining of disk and network bandwidth. Depending on the support for low-priority I/O in the underlying system, reclaim I/O could have some impact on foreground I/O performance. In Everest this tradeoff is controlled by setting the concurrency level of the reclaim process, which limits the number of outstanding reclaim I/Os per Everest client. This is currently a fixed value that is configured per-client: it could also be dynamically regulated using a control process such as MS Manners [8].

3.2.4 N-way off-loading for fault tolerance

Off-loading introduces a failure dependency from a client to the store to which it has off-loaded data. If the store fails, the off-loaded data will become unavailable. To maintain the same availability and persistence as the non off-loading case, Everest masks this failure dependency by adding support for fault-tolerance. Broadly, there are two classes of failure: transient (crash) failures causing temporary data unavailability, and permanent disk failures causing data loss. We first discuss transient failures and then permanent disk failures; this paper does not consider Byzantine failures.

Everest provides fault-tolerance through N-way off-loading. Each off-loaded write is sent to N stores on N different servers, where N is a per-client configurable parameter. In general the client's store set will be larger than N, allowing the client to choose the N least-loaded stores and minimizing the impact on other workloads. Since writes are versioned by Everest, each write can be sent to any N stores independently of previous writes. Note that the client still maintains a consistent view of all the data, and all access to the data is mediated through the client. Hence there is no need for any consensus or co-ordination protocol between the stores.

N-way off-loading can mask up to $N-1$ store failures. If an Everest store fails, the client continues to service requests using the remaining stores. However data held on the failed store now has only $N-1$ copies and is more vulnerable to subsequent failures. The client reclaims such vulnerable data before any other data. The amount of vulnerable data to be reclaimed is bounded by limiting the amount of data off-loaded to any single store.

When an off-loaded version on a store becomes stale, the client normally sends a deletion request to that store. However if the store has failed, the client must ensure that any stale versions on it are eventually deleted. In this case the deletion request is queued locally and persistent, versioned deletion records are written to N other stores in the store set. These ensure that if the client crashes it can correctly reconstruct the outstanding deletions. When the failed store becomes available, the client sends it the queued deletion requests. When these are acknowledged the client garbage-collects the deletion records.

If a volume hosting a store is permanently decommissioned, Everest clients using the store must delete it from their store set. The clients then garbage-collect all deletion records pertaining to that store. Deletion of a store from the clients' store sets can be done manually by the administrator when decommissioning a volume, or automatically by clients when a store has been unresponsive for a certain amount of time.

Permanent disk failures: N-way off-loading also serves to protect against permanent disk failures on stores by adding redundant copies. An alternative approach is to ensure that Everest stores themselves are resilient to disk failures, for example, if the underlying storage uses RAID. In this approach, the system administrator configures each client to use only stores whose base volumes have at least as much redundancy as the client's base volume.

Reading and reclaiming: The Everest client load-balances reads of N-way off-loaded data by sending each read to the least loaded store holding the data. However, read requests for off-loaded data are rare. While reclaiming, low-priority $read\_any$ operations are issued concurrently to multiple stores: the first store to respond sends a cancellation request to the other stores.

3.3 Implementation

The current Everest prototype is implemented at user level. The store is implemented as an RPC server exporting the four store operations. The client is implemented as two library layers. The virtual store layer exports the Everest store operations for the virtual store, implemented by load-balancing across the Everest stores in the store set. The policy layer exports a standard block read/write interface and issues requests to the base volume and to the virtual store layer as appropriate.

The user-level library is sufficient for testing Everest with block-level traces and micro-benchmarks. We are also able to use the user-level prototype with unmodified binary Windows applications such as SQL Server. To do this, we intercept the application's file I/O calls using DLL redirection [13]; a policy layer then maps these calls to virtual store operations.

4 Evaluation

The main aim of the Everest design is to improve I/O performance under peak load. The first part of this section quantifies this improvement by measuring the impact of peak off-loading on I/O response times, using block-level traces of a production Exchange mail server.

While trace replay from a production server gives us a realistic evaluation of I/O response times, application benchmarks let us measure the improvement in end-to-end application throughput. The second part of the evaluation is complementary to the first: it uses an OLTP benchmark with an unmodified SQL Server application to measure the performance of off-loading in various configurations, and identifies the individual performance benefits of off-loading and of using a log-structured store. Finally it shows how application performance scales as more idle spindles are added to the network as well as when more load is added.

Finally, this section uses micro-benchmarks to evaluate the sensitivity of the base Everest performance to the read-write ratio of the workload, since Everest is designed to off-load write load but not read load. It also uses micro-benchmarks to test the scaling of the I/O performance as idle spindles are added, as well as the limits on efficiency of the reclaim process.

All the results in this evaluation are based on experiments run on a hardware testbed running Everest. The testbed consists of four HP servers, each with a dual-core Intel Xeon 3 GHz processor and an HP SmartArray 6400 series RAID controller connected to a rack-mounted disk enclosure with a SCSI backplane. Each enclosure contained 14 high-end enterprise disks: 15K RPM Seagate Cheetahs. The servers were connected to each other by a switched 1 Gbps Ethernet. Communication between Everest clients and stores uses in-process shared memory within a single machine; TCP for unicast communication across machines, and UDP subnet broadcast for periodic broadcasts of store load metrics. For N-way off-loading Everest clients use a combination of TCP and UDP to implement reliable multicast.

4.1 I/O response times: Exchange

In Section 2 we described the I/O traces from a production Exchange mail server, which motivated peak off-loading. Here we evaluate Everest against those traces by replaying them on our testbed.

We selected three episodes from the 24-hour trace shown in Figure 1 that covered the three highest peaks in request rates (measured on a 1 min time scale). The storage capacity of our testbed machines is much lower than that of the original server, which was 7.2 TB across 8 data volumes. Hence for each episode we selected three volumes: the volumes having the maximum, median, and the minimum number of requests during the episode. This corresponds to a configuration where off-loading is restricted to the three chosen volumes, i.e. the three volumes do not off-load to, or accept off-loads from, the other 5 volumes. The three selected volumes were then mapped onto volumes on one of our testbed servers. To achieve the required capacity, each testbed volume was configured as a RAID-0 array with four 146 GB Seagate Cheetah disks; the original production server uses redundant configurations such as RAID-5.

	Time	Requests	Read %	Peak rate
1	22:28–22:58	444345	27%	98206 reqs/s
2	03:16–04:01	5072789	91%	95179 reqs/s
3	10:05–10:35	620519	24%	35568 reqs/s

Table 1: Peak episode traces.

	Mean		99th percentile	
	Original	Testbed	Original	Testbed
1	1.17 s	1.53 s	8.8 s	11.2 s
2	2.08 s	0.06 s	20.0 s	0.6 s
3	0.43 s	0.04 s	3.1 s	0.4 s

Table 2: Peak episode response times.

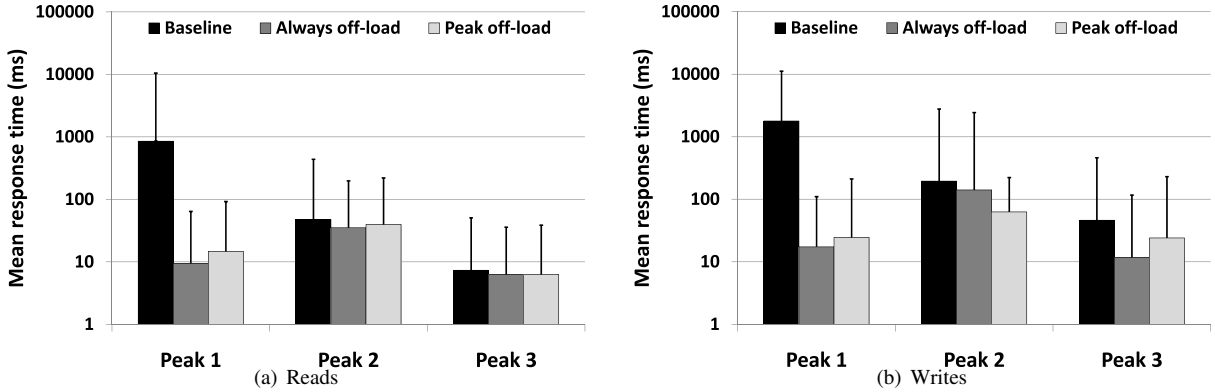
Table 1 shows the three peak episodes covered. The original trace files are divided into 15 min segments; each episode consists of the segment containing the request rate peak, as well as the following segment, to ensure that the trace would be long enough to cover any reclaim activity. Peak episode 2 was extended by an additional 15 min in order to cover all the reclaim activity in all the experiments. Table 2 shows the response times from the original trace as well as those measured on our testbed. The testbed response times are similar to the original hardware for peak 1, and lower (but still high) for the other two peaks.

We evaluated three configurations:

- *Baseline*: no off-load enabled.
- *Always off-load*: all writes are off-loaded, and data is never reclaimed.
- *Peak off-load*: off-load and reclaim are enabled with the queue length based policy described in Section 3.2.2, with $T_{base} = T_{store} = 32$.

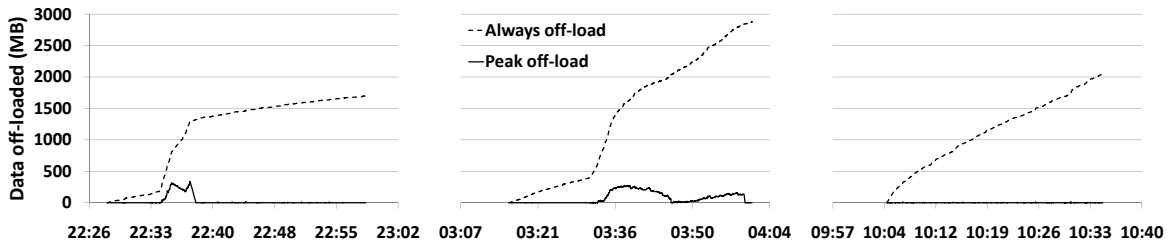
In the two off-load configurations, each volume was configured with an Everest client. Each volume also hosted an Everest store on a small partition (less than 3% of the volume size) at the end of the volume. The off-load configurations used 1-way, single-machine off-loading: for off-loading between RAID volumes on a single server we expect this to be the typical configuration.

Response time: Figure 5 shows the mean and 99th percentile read and write response times achieved by the



(a) Reads (b) Writes

Figure 5: Effect of off-load on mean response time (log scale). Error bars show 99th percentile response times.



(a) Peak 1 (b) Peak 2 (c) Peak 3

Figure 6: Amount of off-loaded data over time.

three configurations for all three peaks. Note that the y-axes are on a log scale. Off-loading improved both read and write response times by almost two orders of magnitude for the worst peak (peak 1), and significantly for the other two peaks. Writes benefit by off-loading from the heavily loaded volume to the lightly loaded ones, and reads benefit by having fewer writes to contend with.

For the write-dominated peaks 1 and 3, "always off-load" has better performance than "peak off-load". This is because it balances load more aggressively than the "peak off-load" policy, which only off-loads when the base volume is heavily loaded and some Everest store is lightly loaded. The disadvantage of "always off-load" is that if the Everest store's underlying disks are busy, then the resulting contention will hurt performance for both contending workloads. This effect can be seen clearly for peak 2, where the write performance of "always off-load" is worse than "peak off-load".

Reclaim time: It is important to know how much data is off-loaded and for how long, since this affects space usage on Everest stores as well as vulnerability to failures. Figure 6 shows for each peak episode the amount of off-loaded data over time, summed across all three clients, for both the "peak off-load" and the "always off-load" policy. With "always off-load", data is off-loaded more aggressively and never reclaimed. Hence, the amount of off-loaded data increases at a much higher rate and never decreases. Interestingly this resulted in more off-loaded data for peak 2 than for the other two; although peak 2 only has 9% writes, in absolute terms it has more writes than the other two peaks. By contrast, the "peak off-load" policy off-loads less data and reclaims it using idle bandwidth: this keeps the amount of off-loaded data low. For peak 3, reclaiming using idle bandwidth is so effective that the amount of off-loaded data is always close to zero. The difference between the two policies shows the importance of off-loading selectively, and of reclaiming off-loaded data.

Sensitivity to parameters: Figure 7 shows the sensitivity of off-load performance to three parameters: T_{base}, the base volume load below which the client will not off-load; T_{store}, the maximum load on a store beyond which clients will not off-load to it; and $N_{reclaim}$, the maximum number of concurrent reclaim I/Os per client. In each case one parameter was varied while the others were set to their default values: $T_{base} = 32$, $T_{store} = 32$, and $N_{reclaim} = 256$. The graphs show the mean response time and the mean amount of off-loaded data, across all clients and all peaks. Performance is generally insensitive to T_{base} and T_{store}: when T_{store} is high, off-loaded I/Os begin to contend with non off-loaded I/Os, but this effect is small compared to the order-of-magnitude re-

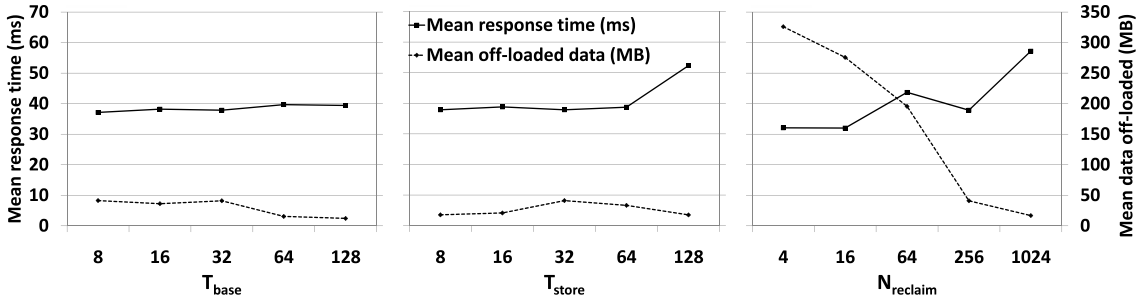
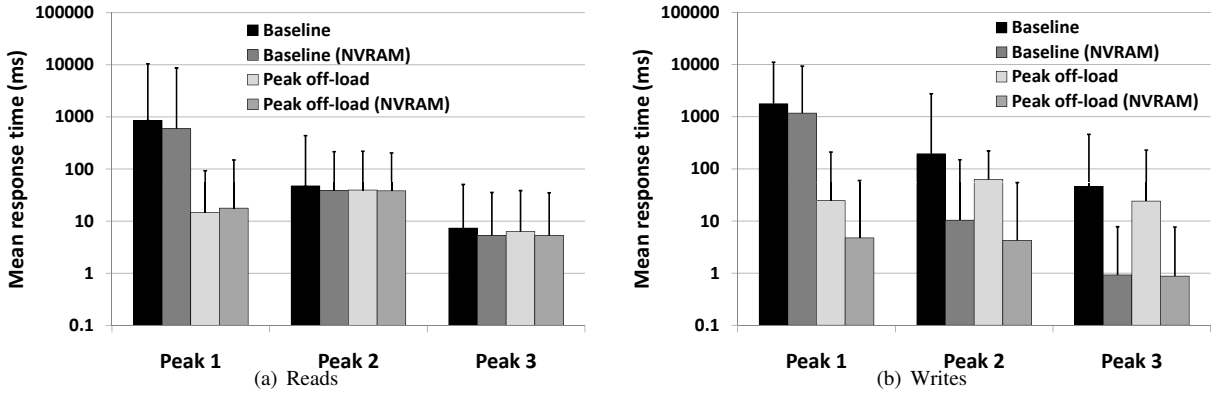


Figure 7: Sensitivity to off-load and reclaim parameters.



(a) Reads (b) Writes

Figure 8: Effect of off-load with NVRAM.

ductions achieved by off-loading. Performance degrades slightly with higher $N_{reclaim}$, due to contention between reclaim I/Os and foreground I/Os. However, for this particular workload, increasing $N_{reclaim}$ from 64 to 256 reduced the fraction of reads that was off-loaded from 0.4% to 0.1%, resulting in a slight decrease in overall response time. The main effect of $N_{reclaim}$ is on the amount of data left off-loaded: decreasing $N_{reclaim}$ below 256 underutilizes the available idle disk bandwidth and it takes substantially longer to reclaim data.

NVRAM: Enterprise RAID controllers are often augmented with battery-backed non-volatile RAM (NVRAM), which can potentially improve write performance by persisting writes temporarily in the NVRAM. The original traced server had 512 MB of NVRAM shared across all its volumes and yet had high response times under peak load. To confirm this, we enabled the NVRAM on our test server. This was 128 MB shared across 3 volumes, set to the default configuration of 50% for read acceleration and 50% for writes. Figure 8 compares the baseline and peak off-load performance with and without NVRAM. Even with NVRAM, the largest peak (peak 1) still shows very high read and write response times, because the amount of data written exceeded the capacity of the NVRAM. Peak off-loading by contrast substantially reduces response times for peak 1.

Max log size	982MB
Max log records	218505
Max valid data	193MB
Max recovery time	14.5 s
Total meta-data (compressed)	247 KB

Table 3: Worst-case recovery statistics.

Overheads: The main CPU overhead of Everest is the lookup and update of in-memory meta-data, which are logarithmic in the size of the meta-data. In our experiments these added an average CPU overhead of 56 μs per off-loaded write, which translated to an average CPU consumption over all 3 peaks of 0.4%. The meta-data also adds a memory overhead; the high watermark of the meta-data memory usage was 11.7 MB. With a more optimized implementation these overheads could be further reduced. Since our testbed was limited to four machines we did not measure the scalability of using subnet broadcast for load updates. However we note that the broadcasts are limited to a subnet, and ethernet multicast could be used to further limit their scope. The broadcast frequency is also low: at most once every 100 ms.

Recovery times: Off-loading occurs during relatively infrequent peak load episodes; hence usually the amount of off-loaded state to be recovered after failure will be

very small. We measured recovery times for Everest clients and stores in a worst-case scenario where the Exchange server fails when the amount of off-loaded data is highest. For our traces, this was 10 min into peak episode 1. Table 3 summarizes the results. Since Everest stores recover concurrently, we show statistics for the store which was the slowest to recover. The recovery time is essentially the time to scan the entire log on disk sequentially. In the single-machine scenario the Everest client can recover as soon as all stores in its store set have recovered. In the network off-load case, the stores must also transfer meta-data over the network to the client; however, as shown in Table 3, the meta-data is small.

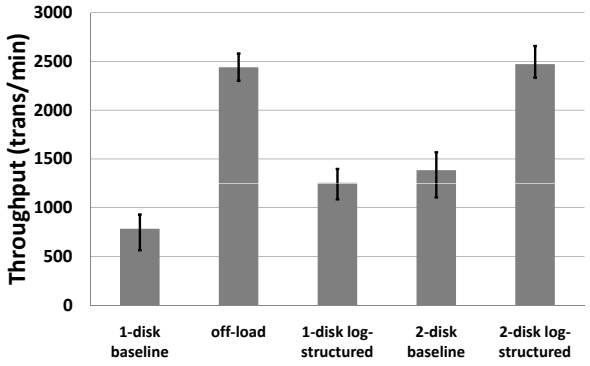


Figure 9: Single-server OLTP throughput.

4.2 Application throughput: SQL Server

The previous section showed substantial improvements in I/O response time from peak off-loading, based on trace replay. In this section we use a standard benchmark to measure the effect of off-loading on the end-to-end throughput of the application. Our test application is Microsoft SQL Server running an OLTP (TPC-C like) benchmark. The figure of merit is the transaction throughput measured as the number of "new order" transactions successfully completed per minute. We measure the saturation throughput, i.e. we increase the concurrency level until the server can support no more connections, and then measure the achieved throughput.

In the OLTP workload 43.5% of the transactions are updates ("new order" transactions), which results in about 27–32% writes at the disk I/O level. The disk workload has poor locality: when the database is larger than the available buffer cache memory, the performance is limited by the random-access I/O performance of the underlying storage. In our experiments we used a database size of 7.5 GB (corresponding to 100 warehouses) and a SQL Server buffer cache size of 256 MB.

The database server used was an unmodified SQL Server 2005 SP2, and we intercepted the server's file I/O using DLL redirection. Following standard practice, the database file and the transaction log file were stored on two separate volumes; each of these contained a single Seagate Cheetah 15K disk. Off-loading was enabled only for the database file, since it was the bottleneck device: we measured the utilization of the log disk and found that it was 7.6% on average and under 20% always. The policy used was the queue length based policy described in Section 3, with $T_{base} = T_{store} = 32$.

All results shown are the average of 5 runs; error bars show the minimum and the maximum values observed. Each experiment was run for 10 min to warm up the cache, and another 10 min to measure the throughput.

In our first experiment, we measured the benefit of off-load for OLTP on a single server. We compared the baseline throughput with no off-loading enabled, to that obtained when off-loading was enabled with a single Everest store on a different disk on the same server. The first two bars in Figure 9 show that off-loading provides a 3x increase in throughput over the baseline. We repeated the same experiment with the Everest store on a different testbed machine to the client; the results were quantitatively the same, indicating that the network was not a bottleneck for this configuration.

Two factors contribute to this benefit: the additional spindle provides additional I/O throughput, and random-access writes are converted to sequential-access writes on the Everest store's log. To separate out the contributions of these two factors, we evaluated two more configurations, shown in the third and fourth bars of Figure 9: a 1-disk log-structured configuration, and a 2-disk striped (RAID-0) array with the default (non-log-structured) layout. Each does roughly 2x better than the baseline case. This shows that for this workload, the benefit of log-structured writes is roughly equal to that of adding a second spindle.

The last bar in Figure 9 shows the performance of a configuration which is log-structured and uses a 2-disk striped array. We see that the performance of off-loading is comparable to that of this configuration. In other words, off-loading gets the full benefit both of log-structured writes and of an additional spindle, but by opportunistically using the second spindle for off-load rather than explicitly provisioning it.

Impact of unreclaimed data: By only off-loading to lightly loaded volumes, Everest ensures that off-loaded writes will not severely degrade the performance of the applications using those volumes. However reads of off-loaded data must be sent to the Everest store that holds the latest version of the data. In the common case, off-loaded data is reclaimed before being read, i.e. foreground reads on the Everest store are rare. Additionally, if N-way off-loading is enabled, the client is likely to find at least one lightly loaded replica to read.

However, in the worst case, there can be contention for

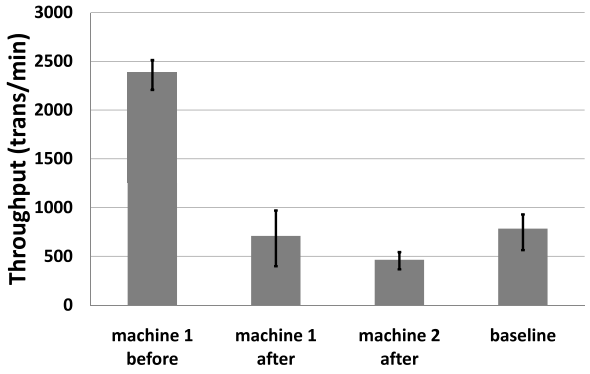


Figure 10: Off-loading worst-case scenario.

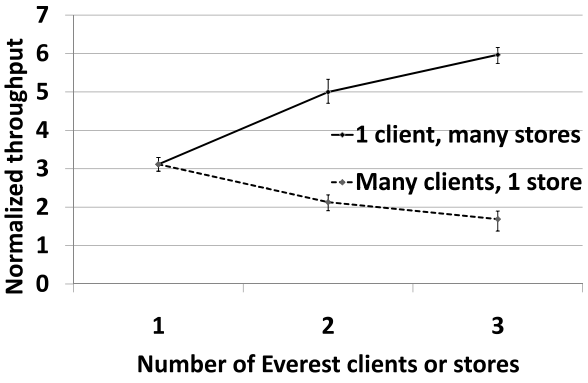


Figure 11: Scaling of OLTP off-load performance.

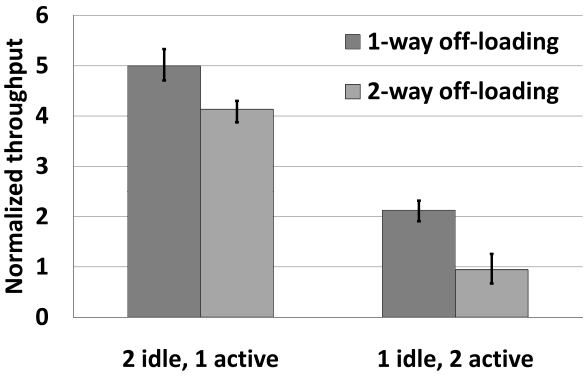


Figure 12: OLTP throughput with 1-way and 2-way off-loading.

disk bandwidth between an Everest store satisfying read requests and applications using the store's base volume. To quantify this effect, we measured a "worst-case" scenario using two machines both running SQL Server and OLTP. Machine 1 was configured for off-load, and machine 2 hosted an Everest store on the same disk as the OLTP database. For the first 10 min of the experiment, only machine 1 was active and it achieved the expected performance improvement. At the end of 10 min, machine 1 had off-loaded 936 MB (13% of the database) to the store on machine 2. At this point machine 2 also became active. Due to this competing load, machine 1 stopped off-loading fresh data onto machine 2; however, accesses to already off-loaded data still went to the Everest store on machine 2. Figure 10 shows the performance of the machines before and after machine 2 became active, compared to the baseline (no off-loading) case. Machine 1's performance dropped to 93% of baseline: relatively few of its I/Os were redirected to the contended disk. Machine 2's performance dropped to 59% of the baseline value: all its I/Os went to the contended disk.

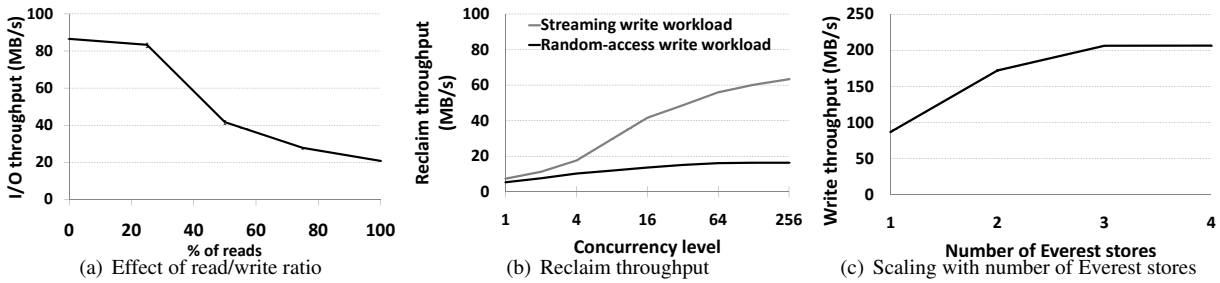
This experiment shows a worst-case scenario of continuous load with no idle periods. Everest is not designed for such scenarios, but rather to improve performance when load is bursty and unbalanced. To avoid off-loading during long periods of high load, the amount of data off-loaded by each client can be bounded. This limits the amount of contention in a worst-case scenario.

Performance scaling: In general, one or more Everest clients can share one or more Everest stores. A single client can improve performance by off-loading to more than one idle store; alternatively multiple clients can off-load to a single idle store.

To measure the effect of using multiple idle stores, we ran the OLTP benchmark on one of the SQL Servers with off-loading enabled, and added 1–3 machines hosting Everest stores but otherwise idle. The solid line in Figure 11 shows the throughput achieved by SQL Server. The throughput is normalized to the baseline throughput of SQL Server with no off-loading (the baseline case is not shown on the graph). With one idle store we get a 3x improvement over the baseline, as seen before. The throughput scales linearly up to 3 idle stores, at which point SQL Server becomes CPU-bound.

To measure the effect of multiple clients sharing a single idle store, we configured one machine to host an Everest store and added 1–3 machines running OLTP and configured for off-load. The dashed line in Figure 11 shows the normalized average throughput achieved by the active machines. As more load is added, per-server throughput decreases slightly, since the benefit of the additional spindle must be shared. However, even with 3 SQL Servers sharing a single Everest store, we get an average speedup of 1.7x.

N-way off-loading: Everest supports N-way off-loading to create redundant copies for fault-tolerance. In general this adds overheads due to the bandwidth used to write additional copies. We compared the performance of 1-way and 2-way off-loading on a configuration with one machine running OLTP and two idle machines hosting Everest stores. The throughput of these two cases, normalized to the 1-disk baseline throughput, is shown by the first two bars in Figure 12. We see that 2-way



(a) Effect of read/write ratio (b) Reclaim throughput (c) Scaling with number of Everest stores

Figure 13: Micro-benchmark results.

off-loading incurs a performance penalty since it uses twice as much write bandwidth on the stores; however the penalty is relatively small since the stores are write-optimized. The second pair of bars in Figure 12 represents a configuration with two active SQL Servers and one idle machine. In this case, with 1-way off-loading the two clients share the performance benefit of off-loading to one extra spindle. With 2-way replication we get performance equivalent to the baseline. This is because neither Everest client can find two lightly loaded Everest stores and hence no off-loading occurs.

We also see that 2-way off-load to two stores does better than 1-way off-load to one store; this is because although the write load per store is the same, the read load is halved in the former case. However, in general when using Everest for short-term peak off-loading, we would not expect a high read load on the Everest stores: the main purpose of N-way off-loading is to provide fault-tolerance rather than read performance.

4.3 Micro-benchmarks

In this section we use synthetic micro-benchmarks to measure the sensitivity of Everest performance to the workload read/write ratio and the reclaim concurrency level, as well as the scaling of performance with the number of clients and stores. The results presented here all use 64 KB reads and writes, and are based on the average of 5 runs of each experiment, with error bars showing the maximum and minimum values. Each experiment runs for 20 seconds. All configurations used a single Everest client with a single disk, and one or more Everest stores each with a single disk. The Everest client is configured with an "always off-load" policy.

Figure 13(a) shows the throughput achieved by an Everest client with a single store for a random-access workload, as a function of the workload's read/write ratio. With a write-only workload, Everest achieves a throughput equal to the streaming write bandwidth of the disk (observed to be 90 MB/s), since it converts random-access writes to sequential accesses. As the fraction of reads increases, the throughput is increasingly dominated by the reads: since the workload has no temporal locality, almost all reads go to the base disk (observed random-access throughput for this disk type is 21 MB/s). However, off-loading continues to show benefit until the workload is 100% read-dominated.

Figure 13(b) shows the reclaim throughput achieved by Everest with an otherwise unloaded base disk and Everest store, for random-access and streaming write workloads. Since off-loaded records are reclaimed in log order, the writes to the base disk during reclaim have the same locality pattern as the original write workload. For a random-access workload, reclaim throughput is dominated by the random-access write performance of the base disk, which increases with the concurrency level to a maximum of about 20 MB/s. For a streaming workload, the throughput is limited by the Everest store, which alternates between reading from the tail of the log and writing deletion records to the head. At higher concurrency levels, this effect is amortized by batching, and we can achieve a throughput of up to 70% of the streaming bandwidth of the disk.

We conclude the evaluation section by measuring the scaling of client write throughput as a function of the number of available stores. The results are shown in Figure 13(c). Scaling is linear up to 2 stores. With 3 stores the system becomes CPU-bound, achieving 206 MB/s, or 3300 IOPS. About 9% of the CPU time was spent in Everest meta-data operations; the remainder was due to our user-level trace replay infrastructure, which requires at least one user-kernel transition and two memory copies per I/O.

5 Related work

Most research on storage system performance has focused on achieving good performance when I/O loads remain within expected limits. In contrast, Everest mitigates performance degradations from short-term unpredictable bursts, and is orthogonal to these efforts. We are unaware of much work that exploits pooled storage as a short term store to improve performance.

Self-scaling systems and data migration: There has been much work on dynamic reconfiguration of storage to adapt to workloads, e.g. switching between RAID-5 and mirroring [24] or re-encoding of data in cluster storage [1]. Many storage systems [2, 5, 7, 10, 11, 14, 18] scale incrementally by redistributing existing data across newly added nodes; the same mechanisms can also be used to dynamically reconfigure individual workloads when load changes. For example, Amazon's Dynamo [7] can redistribute keys across nodes to balance load; however this is a heavyweight operation not designed for use when one node becomes a temporary I/O hotspot.

In general data migration allows a storage system to adapt to changes in workload behavior over the long term. It is not suitable for short, upredictable bursts of load where the system cannot predict ahead of time when, how, and what data to migrate. Migrating or re-encoding data during the load burst itself will add additional load to the system. For short unpredictable bursts a better approach is to opportunistically and temporarily balance the load with minimal migration of data, as is done in Everest. Data is not permanently migrated or re-encoded in Everest, but temporarily off-loaded and then reclaimed to the original volume. It is thus complementary to long-term reconfiguration through migration or re-encoding of data. Everest is also transparent to applications and file systems, and hence can be incrementally deployed with minimal changes to server software.

Automatic provisioning: Provisioning tools such as HP's Disk Array Designer [4] generate storage system configurations from workload characteristics and service-level objectives. Such tools can optimize for different goals, such as cost, performance, or power [21]. However workload characteristics must be explicitly specified, perhaps using a language such as Rome [23].

Workloads with high peak-to-mean ratios are a problem for these approaches. Even if the peak levels are known, the only option to get good performance at peak load is to massively over-provision the storage subsystem. This problem is compounded by the rapid increase in disk capacity compared to throughput over time: increasingly throughput is driving the number of disks required in data volumes. Rather than provisioning each data volume for its peak load, Everest exploits the fact that peaks may not be correlated among all the workloads in a data center, and statistically multiplexes storage across workloads for short periods of time. This allows volumes to be provisioned for common-case loads or 99th percentiles rather than worst-case peaks.

Write off-loading: In previous work [15] we showed that write off-loading can save energy by increasing the length of idle periods and allowing disks to spin down. In the current work we re-use much of the same infrastructure — with the addition of N-way replication for fault-tolerance — to address performance degradation due to unbalanced peak loads. The two applications of write off-loading are complementary, and exploit different workload characteristics to achieve different ends.

Log-structured stores: Everest stores are log-structured, in order to be able to efficiently handle a write dominated workload. They are different in design and use from traditional log-structured file systems [17, 19] as well as file system journals, DBMS transaction logs, or block-level write-ahead logs [6]. Unlike a log-structured file system, data is only stored in Everest stores for short periods of time, and in the common case stores do not serve application reads. Free space on the store is created by clients reclaiming and deleting data, rather than by using a log cleaner. Write-ahead logs and journals are used to improve write response times; however they share the same disk resources as the underlying DBMS or file system. By contrast, Everest clients opportunistically use idle bandwidth on other volumes to off-load writes.

Idle disk bandwidth: Everest utilizes idle disk bandwidth for off-load and reclaim. Modern storage subsystems might also use idle disk bandwidth for a variety of background maintenance tasks [12, 22].

Solid-state storage: There have been many proposals for using flash-based solid state memory for storage [3, 9, 16, 25]. This might be in the form of solid-state storage devices (SSDs), or the flash might be added inside the disks, in the RAID controllers, or on the motherboards. The Everest store design could be used with any of these. The circular log design results in writes being sequential and evenly spread over the flash memory, and hence is optimal both for write performance and for wear-leveling [9].

6 Conclusion

Server I/O workloads are bursty, and under peak load performance can degrade significantly. Everest addresses this by pooling idle disk bandwidth into a virtual short-term persistent store to which overloaded volumes can off-load write requests. These volumes can then dedicate their I/O bandwidth to read requests, thereby improving performance. Everest can be interposed transparently at the block I/O level, and unmodified applications can benefit from its use. We have demonstrated the effectiveness of the approach using traces from a production Exchange server, as well as benchmarks.

Acknowledgements: We thank Bruce Worthington, Swaroop Kavalanekar and Chris Mitchell for the production Exchange server traces used in this paper. We also thank our shepherd Kim Keeton and the anonymous reviewers for their feedback.

References

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: versatile cluster-based storage. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, Dec. 2005.

[2] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2008.

[4] E. Anderson, S. Spence, R. Swaminathan, M. Kallahalla, and Q. Wang. Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems*, 23(4):337–374, November 2005.

[5] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, Dec. 1995.

[6] W. J. Bolosky. Improving disk write performance by logging. Private Communication.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[8] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, SC, Dec. 1999.

[9] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *CSURV: Computing Surveys*, 37, 2005.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *ACM Symposium on Operating System Principles*, Lake George, NY, Oct. 2003.

[11] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, Oct. 1998.

[12] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. In *Proc. USENIX Annual Technical Conference*, New Orleans, LA, Jan. 1995.

[13] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proc. USENIX Windows NT Symposium*, Seattle, WA, July 1999.

[14] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, Oct. 1996.

[15] D. Narayanan, A. Donnelly, and A. Rowstron. Write offloading: Practical power management for enterprise storage. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, Feb. 2008.

[16] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional flash. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[17] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, Pacific Grove, CA, Oct. 1991.

[18] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. FAB: building distributed enterprise disk arrays from commodity components. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, Oct. 2004.

[19] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proc. USENIX Winter Conference*, San Diego, CA, Jan. 1993.

[20] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[21] J. D. Strunk, E. Thereska, C. Faloutsos, and G. R. Ganger. Using utility to provision storage systems. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, Feb. 2008.

[22] E. Thereska, J. Schindler, J. Bucy, B. Salmon, C. R. Lumb, and G. R. Ganger. A framework for building unobtrusive disk maintenance applications. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, Mar. 2004.

[23] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. In *Proc. International Workshop on Quality of Service (IWQoS)*, Karlsruhe, Germany, June 2001.

[24] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[25] D. Woodhouse. JFFS: The journalling flash file system. In *Proc. The Linux Symposium*, Ottawa, Canada, July 2001.

[26] B. Worthington and S. Kavalanekar. Characterization of storage workload traces from production Windows servers. In *Proc. IEEE International Symposium on Workload Characterization (IISWC)*, Seattle, WA, Sept. 2008.

Improving MapReduce Performance in Heterogeneous Environments

Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, Ion Stoica
University of California, Berkeley
{matei,andyk,adj,randy,stoica}@cs.berkeley.edu

Abstract

MapReduce is emerging as an important programming model for large-scale data-parallel applications such as web indexing, data mining, and scientific simulation. Hadoop is an open-source implementation of MapReduce enjoying wide adoption and is often used for short jobs where low response time is critical. Hadoop's performance is closely tied to its task scheduler, which implicitly assumes that cluster nodes are homogeneous and tasks make progress linearly, and uses these assumptions to decide when to speculatively re-execute tasks that appear to be stragglers. In practice, the homogeneity assumptions do not always hold. An especially compelling setting where this occurs is a virtualized data center, such as Amazon's Elastic Compute Cloud (EC2). We show that Hadoop's scheduler can cause severe performance degradation in heterogeneous environments. We design a new scheduling algorithm, Longest Approximate Time to End (LATE), that is highly robust to heterogeneity. LATE can improve Hadoop response times by a factor of 2 in clusters of 200 virtual machines on EC2.

1 Introduction

Today's most popular computer applications are Internet services with millions of users. The sheer volume of data that these services work with has led to interest in parallel processing on commodity clusters. The leading example is Google, which uses its MapReduce framework to process 20 petabytes of data per day [1]. Other Internet services, such as e-commerce websites and social networks, also cope with enormous volumes of data. These services generate clickstream data from millions of users every day, which is a potential gold mine for understanding access patterns and increasing ad revenue. Furthermore, for each user action, a web application generates one or two orders of magnitude more data in system logs, which are the main resource that developers and operators have for diagnosing problems in production.

The MapReduce model popularized by Google is very attractive for ad-hoc parallel processing of arbitrary data. MapReduce breaks a computation into small tasks that run in parallel on multiple machines, and scales easily to very large clusters of inexpensive commodity computers. Its popular open-source implementation, Hadoop [2], was developed primarily by Yahoo, where it runs jobs that produce hundreds of terabytes of data on at least 10,000 cores [4]. Hadoop is also used at Facebook, Amazon, and Last.fm [5]. In addition, researchers at Cornell, Carnegie Mellon, University of Maryland and PARC are starting to use Hadoop for seismic simulation, natural language processing, and mining web data [5, 6].

A key benefit of MapReduce is that it automatically handles failures, hiding the complexity of fault-tolerance from the programmer. If a node crashes, MapReduce reruns its tasks on a different machine. Equally importantly, if a node is available but is performing poorly, a condition that we call a *straggler*, MapReduce runs a *speculative copy* of its task (also called a "backup task") on another machine to finish the computation faster. Without this mechanism of *speculative execution*[1], a job would be as slow as the misbehaving task. Stragglers can arise for many reasons, including faulty hardware and misconfiguration. Google has noted that speculative execution can improve job response times by 44% [1].

In this work, we address the problem of how to robustly perform speculative execution to maximize performance. Hadoop's scheduler starts speculative tasks based on a simple heuristic comparing each task's progress to the average progress. Although this heuristic works well in homogeneous environments where stragglers are obvious, we show that it can lead to severe performance degradation when its underlying assumptions are broken. We design an improved scheduling algorithm that reduces Hadoop's response time by a factor of 2.

An especially compelling environment where

[1] Not to be confused with speculative execution at the OS or hardware level for branch prediction, as in Speculator [11].

Hadoop's scheduler is inadequate is a virtualized data center. Virtualized "utility computing" environments, such as Amazon's Elastic Compute Cloud (EC2) [3], are becoming an important tool for organizations that must process large amounts of data, because large numbers of virtual machines can be rented by the hour at lower costs than operating a data center year-round (EC2's current cost is $0.10 per CPU hour). For example, the New York Times rented 100 virtual machines for a day to convert 11 million scanned articles to PDFs [7]. Utility computing environments provide an economic advantage (paying by the hour), but they come with the caveat of having to run on virtualized resources with uncontrollable variations in performance. We also expect heterogeneous environments to become common in private data centers, as organizations often own multiple generations of hardware, and data centers are starting to use virtualization to simplify management and consolidate servers. We observed that Hadoop's homogeneity assumptions lead to incorrect and often excessive speculative execution in heterogeneous environments, and can even degrade performance below that obtained with speculation disabled. In some experiments, as many as 80% of tasks were speculatively executed.

Naïvely, one might expect speculative execution to be a simple matter of duplicating tasks that are sufficiently slow. In reality, it is a complex issue for several reasons. First, speculative tasks are not free – they compete for certain resources, such as the network, with other running tasks. Second, choosing the node to run a speculative task on is as important as choosing the task. Third, in a heterogeneous environment, it may be difficult to distinguish between nodes that are slightly slower than the mean and stragglers. Finally, stragglers should be identified as early as possible to reduce response times.

Starting from first principles, we design a simple algorithm for speculative execution that is robust to heterogeneity and highly effective in practice. We call our algorithm LATE for Longest Approximate Time to End. LATE is based on three principles: prioritizing tasks to speculate, selecting fast nodes to run on, and capping speculative tasks to prevent thrashing. We show that LATE can improve the response time of MapReduce jobs by a factor of 2 in large clusters on EC2.

This paper is organized as follows. Section 2 describes Hadoop's scheduler and the assumptions it makes. Section 3 shows how these assumptions break in heterogeneous environments. Section 4 introduces our new scheduler, LATE. Section 5 validates our claims about heterogeneity in virtualized environments through measurements of EC2 and evaluates LATE in several settings. Section 6 is a discussion. Section 7 presents related work. Finally, we conclude in Section 8.

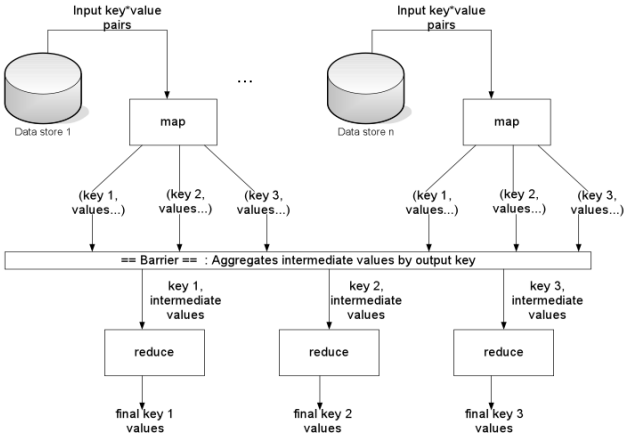


Figure 1: A MapReduce computation. Image from [8].

2 Background: Scheduling in Hadoop

In this section, we describe the mechanism used by Hadoop to distribute work across a cluster. We identify assumptions made by the scheduler that hurt its performance. These motivate our LATE scheduler, which can outperform Hadoop's by a factor of 2.

Hadoop's implementation of MapReduce closely resembles Google's [1]. There is a single *master* managing a number of *slaves*. The input file, which resides on a distributed filesystem throughout the cluster, is split into even-sized *chunks* replicated for fault-tolerance. Hadoop divides each MapReduce job into a set of *tasks*. Each chunk of input is first processed by a *map* task, which outputs a list of key-value pairs generated by a user-defined map function. Map outputs are split into buckets based on key. When all maps have finished, *reduce* tasks apply a reduce function to the list of map outputs with each key. Figure 1 illustrates a MapReduce computation.

Hadoop runs several maps and reduces concurrently on each slave – two of each by default – to overlap computation and I/O. Each slave tells the master when it has empty task slots. The scheduler then assigns it tasks.

The goal of speculative execution is to minimize a job's *response time*. Response time is most important for short jobs where a user wants an answer quickly, such as queries on log data for debugging, monitoring and business intelligence. Short jobs are a major use case for MapReduce. For example, the average MapReduce job at Google in September 2007 took 395 seconds [1]. Systems designed for SQL-like queries on top of MapReduce, such as Sawzall [9] and Pig [10], underline the importance of MapReduce for ad-hoc queries. Response time is also clearly important in a pay-by-the-hour environment like EC2. Speculative execution is less useful in long jobs, because only the last wave of tasks is affected, and it may be inappropriate for batch jobs if throughput is

the only metric of interest, because speculative tasks imply wasted work. However, even in pure throughput systems, speculation may be beneficial to prevent the prolonged life of many concurrent jobs all suffering from straggler tasks. Such nearly complete jobs occupy resources on the master and disk space for map outputs on the slaves until they terminate. Nonetheless, in our work, we focus on improving response time for short jobs.

2.1 Speculative Execution in Hadoop

When a node has an empty task slot, Hadoop chooses a task for it from one of three categories. First, any failed tasks are given highest priority. This is done to detect when a task fails repeatedly due to a bug and stop the job. Second, non-running tasks are considered. For maps, tasks with data local to the node are chosen first. Finally, Hadoop looks for a task to execute speculatively.

To select speculative tasks, Hadoop monitors task progress using a *progress score* between 0 and 1. For a map, the progress score is the fraction of input data read. For a reduce task, the execution is divided into three phases, each of which accounts for 1/3 of the score:

- The *copy phase*, when the task fetches map outputs.
- The *sort* phase, when map outputs are sorted by key.
- The *reduce* phase, when a user-defined function is applied to the list of map outputs with each key.

In each phase, the score is the fraction of data processed. For example, a task halfway through the copy phase has a progress score of $\frac{1}{2} \cdot \frac{1}{3} = \frac{1}{6}$, while a task halfway through the reduce phase scores $\frac{1}{3} + \frac{1}{3} + (\frac{1}{2} \cdot \frac{1}{3}) = \frac{5}{6}$.

Hadoop looks at the average progress score of each category of tasks (maps and reduces) to define a *threshold* for speculative execution: When a task's progress score is less than the average for its category minus 0.2, and the task has run for at least one minute, it is marked as a straggler. All tasks beyond the threshold are considered "equally slow," and ties between them are broken by data locality. The scheduler also ensures that at most one speculative copy of each task is running at a time.

Although a metric like progress rate would make more sense than absolute progress for identifying stragglers, the threshold in Hadoop works reasonably well in homogenous environments because tasks tend to start and finish in "waves" at roughly the same times and speculation only starts when the last wave is running.

Finally, when running multiple jobs, Hadoop uses a FIFO discipline where the earliest submitted job is asked for a task to run, then the second, etc. There is also a priority system for putting jobs into higher-priority queues.

2.2 Assumptions in Hadoop's Scheduler

Hadoop's scheduler makes several implicit assumptions:

1. Nodes can perform work at roughly the same rate.
2. Tasks progress at a constant rate throughout time.
3. There is no cost to launching a speculative task on a node that would otherwise have an idle slot.
4. A task's progress score is representative of fraction of its total work that it has done. Specifically, in a reduce task, the copy, sort and reduce phases each take about 1/3 of the total time.
5. Tasks tend to finish in waves, so a task with a low progress score is likely a straggler.
6. Tasks in the same category (map or reduce) require roughly the same amount of work.

As we shall see, assumptions 1 and 2 break down in a virtualized data center due to heterogeneity. Assumptions 3, 4 and 5 can break down in a homogeneous data center as well, and may cause Hadoop to perform poorly there too. In fact, Yahoo disables speculative execution on some jobs because it degrades performance, and monitors faulty machines through other means. Facebook disables speculation for reduce tasks [14].

Assumption 6 is inherent in the MapReduce paradigm, so we do not address it in this paper. Tasks in MapReduce should be small, otherwise a single large task will slow down the entire job. In a well-behaved MapReduce job, the separation of input into equal chunks and the division of the key space among reducers ensures roughly equal amounts of work. If this is not the case, then launching a few extra speculative tasks is not harmful as long as obvious stragglers are also detected.

3 How the Assumptions Break Down

3.1 Heterogeneity

The first two assumptions in Section 2.2 are about homogeneity: Hadoop assumes that any detectably slow node is faulty. However, nodes can be slow for other reasons. In a non-virtualized data center, there may be multiple generations of hardware. In a virtualized data center where multiple virtual machines run on each physical host, such as Amazon EC2, co-location of VMs may cause heterogeneity. Although virtualization isolates CPU and memory performance, VMs compete for disk and network bandwidth. In EC2, co-located VMs use a host's full bandwidth when there is no contention and share bandwidth fairly when there is contention [12]. Contention can come from other users' VMs, in which case it may be transient, or from a user's *own* VMs if they do similar work, as in Hadoop. In Section 5.1, we

measure performance differences of 2.5x caused by contention. Note that EC2's bandwidth sharing policy is not inherently harmful – it means that a physical host's I/O bandwidth can be fully utilized even when some VMs do not need it – but it causes problems in Hadoop.

Heterogeneity seriously impacts Hadoop's scheduler. Because the scheduler uses a fixed threshold for selecting tasks to speculate, *too many* speculative tasks may be launched, taking away resources from useful tasks (assumption 3 is also untrue). Also, because the scheduler ranks candidates by locality, the *wrong* tasks may be chosen for speculation first. For example, if the average progress was 70% and there was a 2x slower task at 35% progress and a 10x slower task at 7% progress, then the 2x slower task might be speculated before the 10x slower task if its input data was available on an idle node.

We note that EC2 also provides "large" and "extra large" VM sizes that have lower variance in I/O performance than the default "small" VMs, possibly because they fully own a disk. However, small VMs can achieve higher I/O performance per dollar because they use all available disk bandwidth when no other VMs on the host are using it. Larger VMs also still compete for network bandwidth. Therefore, we focus on optimizing Hadoop on "small" VMs to get the best performance per dollar.

3.2 Other Assumptions

Assumptions 3, 4 and 5 in Section 2.2 are broken on both homogeneous and heterogeneous clusters, and can lead to a variety of failure modes.

Assumption 3, that speculating tasks on idle nodes costs nothing, breaks down when resources are shared. For example, the network is a bottleneck shared resource in large MapReduce jobs. Also, speculative tasks may compete for disk I/O in I/O-bound jobs. Finally, when multiple jobs are submitted, needless speculation reduces throughput without improving response time by occupying nodes that could be running the next job.

Assumption 4, that a task's progress score is approximately equal to its percent completion, can cause incorrect speculation of reducers. In a typical MapReduce job, the copy phase of reduce tasks is the slowest, because it involves all-pairs communication over the network. Tasks quickly complete the other two phases once they have all map outputs. However, the copy phase counts for only 1/3 of the progress score. Thus, soon after the first few reducers in a job finish the copy phase, their progress goes from 1/3 to 1, greatly increasing the average progress. As soon as about 30% of reducers finish, the average progress is roughly $0.3 \cdot 1 + 0.7 \cdot 1/3 \approx 53\%$, and now *all* reducers still in the copy phase will be 20% behind the average, and an arbitrary set will be speculatively executed. Task slots will fill up, and true stragglers may never be speculated executed, while the network will be overloaded with unnecessary copying. We observed this behavior in 900-node runs on EC2, where 80% of reducers were speculated.

Assumption 5, that progress *score* is a good proxy for progress *rate* because tasks begin at roughly the same time, can also be wrong. The number of reducers in a Hadoop job is typically chosen small enough so that they they can all start running right away, to copy data while maps run. However, there are potentially tens of mappers per node, one for each data chunk. The mappers tend to run in waves. Even in a homogeneous environment, these waves get more spread out over time due to variance adding up, so in a long enough job, tasks from different generations will be running concurrently. In this case, Hadoop will speculatively execute new, fast tasks instead of old, slow tasks that have more total progress.

Finally, the 20% progress difference threshold used by Hadoop's scheduler means that tasks with more than 80% progress can *never* be speculatively executed, because average progress can never exceed 100%.

4 The LATE Scheduler

We have designed a new speculative task scheduler by starting from first principles and adding features needed to behave well in a real environment.

The primary insight behind our algorithm is as follows: We always speculatively execute the task that we think will finish *farthest into the future*, because this task provides the greatest opportunity for a speculative copy to overtake the original and reduce the job's response time. We explain how we estimate a task's finish time based on progress score below. We call our strategy LATE, for Longest Approximate Time to End. Intuitively, this greedy policy would be optimal if nodes ran at consistent speeds and if there was no cost to launching a speculative task on an otherwise idle node.

Different methods for estimating time left can be plugged into LATE. We currently use a simple heuristic that we found to work well in practice: We estimate the *progress rate* of each task as $ProgressScore/T$, where T is the amount of time the task has been running for, and then estimate the time to completion as $(1 - ProgressScore)/ProgressRate$. This assumes that tasks make progress at a roughly constant rate. There are cases where this heuristic can fail, which we describe later, but it is effective in typical Hadoop jobs.

To really get the best chance of beating the original task with the speculative task, we should also only launch speculative tasks on *fast nodes* – not stragglers. We do this through a simple heuristic – don't launch speculative tasks on nodes that are below some threshold, *SlowNodeThreshold*, of total work performed (sum of progress

scores for all succeeded and in-progress tasks on the node). This heuristic leads to better performance than assigning a speculative task to the first available node. Another option would be to allow more than one speculative copy of each task, but this wastes resources needlessly.

Finally, to handle the fact that speculative tasks cost resources, we augment the algorithm with two heuristics:

- A cap on the number of speculative tasks that can be running at once, which we denote *SpeculativeCap*.
- A *SlowTaskThreshold* that a task's progress rate is compared with to determine whether it is "slow enough" to be speculated upon. This prevents needless speculation when only fast tasks are running.

In summary, the LATE algorithm works as follows:

- If a node asks for a new task and there are fewer than *SpeculativeCap* speculative tasks running:
  - Ignore the request if the node's total progress is below *SlowNodeThreshold*.
  - Rank currently running tasks that are not currently being speculated by estimated time left.
  - Launch a copy of the highest-ranked task with progress rate below *SlowTaskThreshold*.

Like Hadoop's scheduler, we also wait until a task has run for 1 minute before evaluating it for speculation.

In practice, we have found that a good choice for the three parameters to LATE are to set the *SpeculativeCap* to 10% of available task slots and set the *SlowNodeThreshold* and *SlowTaskThreshold* to the 25th percentile of node progress and task progress rates respectively. We use these values in our evaluation. We have performed a sensitivity analysis in Section 5.4 to show that a wide range of thresholds perform well.

Finally, we note that unlike Hadoop's scheduler, LATE does not take into account data locality for launching speculative map tasks, although this is a potential extension. We assume that because most maps are data-local, network utilization during the map phase is low, so it is fine to launch a speculative task on a fast node that does not have a local copy of the data. Locality statistics available in Hadoop validate this assumption.

4.1 Advantages of LATE

The LATE algorithm has several advantages. First, it is robust to node heterogeneity, because it will relaunch only the slowest tasks, and only a small number of tasks. LATE prioritizes among the slow tasks based on how much they hurt job response time. LATE also caps the number of speculative tasks to limit contention for shared resources. In contrast, Hadoop's native scheduler has a fixed threshold, beyond which all tasks that are "slow enough" have an equal chance of being launched. This fixed threshold can cause excessively many tasks to be speculated upon.

Second, LATE takes into account node heterogeneity when deciding *where* to run speculative tasks. In contrast, Hadoop's native scheduler assumes that any node that finishes a task and asks for a new one is likely to be a fast node, i.e. that slow nodes will never finish their original tasks and so will never be candidates for running speculative tasks. This is clearly untrue when some nodes are only slightly (2-3x) slower than the mean.

Finally, by focusing on estimated time left rather than progress rate, LATE speculatively executes only tasks that will improve job response time, rather than any slow tasks. For example, if task A is 5x slower than the mean but has 90% progress, and task B is 2x slower than the mean but is only at 10% progress, then task B will be chosen for speculation first, even though it is has a higher *progress rate*, because it hurts the *response time* more. LATE allows the slow nodes in the cluster to be utilized as long as this does not hurt response time. In contrast, a progress rate based scheduler would always re-execute tasks from slow nodes, wasting time spent by the backup task if the original finishes faster. The use of estimated time left also allows LATE to avoid assumption 4 in Section 2.2 (that progress score is linearly correlated with percent completion): it does not matter how the progress score is calculated, as long as it can be used to estimate the finishing order of tasks.

As a concrete example of how LATE improves over Hadoop's scheduler, consider the reduce example in Section 3.2, where assumption 4 (progress score $\approx$ fraction of work complete) is violated and all reducers in the copy phase fall below the speculation threshold as soon as a few reducers finish. Hadoop's native scheduler would speculate arbitrary reduces, missing true stragglers and potentially starting too many speculative tasks. In contrast, LATE would first start speculating the reducers with the slowest copy phase, which are probably the true stragglers, and would stop launching speculative tasks once it has reached the *SpeculativeCap*, avoiding overloading the network.

4.2 Estimating Finish Times

At the start of Section 4, we said that we estimate the time left for a task based on the progress score provided by Hadoop, as $(1 - ProgressScore)/ProgressRate$. Although this heuristic works well in practice, we wish to point out that there are situations in which it can backfire, and the heuristic might incorrectly estimate that a task which was launched *later* than an identical task will finish *earlier*. Because these situations do not occur in typical MapReduce jobs (as explained below), we have

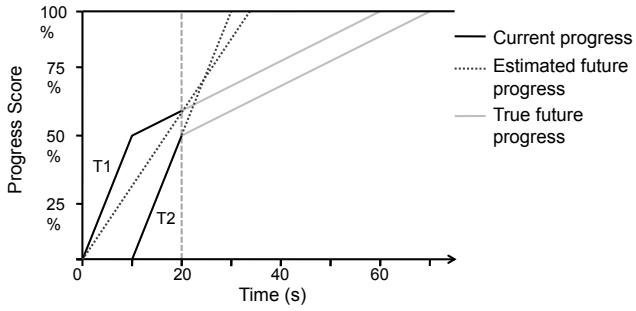


Figure 2: A scenario where LATE estimates task finish orders incorrectly.

used the simple heuristic presented above in our experiments in this paper. We explain this misestimation here because it is an interesting, subtle problem in scheduling using progress rates. In future work, we plan to evaluate more sophisticated methods of estimating finish times.

To see how the progress rate heuristic might backfire, consider a task that has two phases in which it runs at different rates. Suppose the task's progress score grows by 5% per second in the first phase, up to a total score of 50%, and then slows down to 1% per second in the second phase. The task spends 10 seconds in the first phase and 50 seconds in the second phase, or 60s in total. Now suppose that we launch two copies of the task, T1 and T2, one at time 0 and one at time 10, and that we check their progress rates at time 20. Figure 2 illustrates this scenario. At time 20, T1 will have finished its first phase and be one fifth through its second phase, so its progress score will be 60%, and its progress rate will be $60\%/20\text{s} = 3\%/\text{s}$. Meanwhile, T2 will have just finished its first phase, so its progress rate will be $50\%/10\text{s} = 5\%/\text{s}$. The estimated time left for T1 will be $(100\% - 60\%)/(3\%/\text{s}) = 13.3\text{s}$. The estimated time left for T2 will be $(100\% - 50\%)/(5\%/\text{s}) = 10\text{s}$. Therefore our heuristic will say that T1 will take longer to run than T2, while in reality T2 finishes second.

This situation arises because the task's progress rate slows down throughout its lifetime and is not linearly related to actual progress. In fact, if the task *sped up* in its second phase instead of slowing down, there would be no problem – we would correctly estimate that tasks in their first phase have a longer amount of time left, so the estimated *order* of finish times would be correct, but we would be wrong about the exact amount of time left. The problem in this example is that the task slows down in its second phase, so "younger" tasks seem faster.

Fortunately, this situation does not frequently arise in typical MapReduce jobs in Hadoop. A map task's progress is based on the number of records it has processed, so its progress is always representative of percent complete. Reduce tasks are typically slowest in their first phase – the copy phase, where they must read all map outputs over the network – so they fall into the "speeding up over time" category above.

For the less typical MapReduce jobs where some of the later phases of a reduce task are slower than the first, it would be possible to design a more complex heuristic. Such a heuristic would account for each phase independently when estimating completion time. It would use the the per-phase progress rate thus far observed for any completed or in-progress phases for that task, and for phases that the task has not entered yet, it would use the average progress rate of those phases from other reduce tasks. This more complex heuristic assumes that a task which performs slowly in some phases relative to other tasks will not perform relatively fast in other phases. One issue for this phase-aware heuristic is that it depends on historical averages of per phase task progress rates. However, since speculative tasks are not launched until at least the end of at least one wave of tasks, a sufficient number of tasks will have completed in time for the first speculative task to use the average per phase progress rates. We have not implemented this improved heuristic to keep our algorithm simple. We plan to investigate finish time estimation in more detail in future work.

5 Evaluation

We began our evaluation by measuring the effect of contention on performance in EC2, to validate our claims that contention causes heterogeneity. We then evaluated LATE performance in two environments: large clusters on EC2, and a local virtualized testbed. Lastly, we performed a sensitivity analysis of the parameters in LATE.

Throughout our evaluation, we used a number of different environments. We began our evaluation by measuring heterogeneity in the production environment on EC2. However, we were assigned by Amazon to a separate test cluster when we ran our scheduling experiments. Amazon moved us to this test cluster because our experiments were exposing a scalability bug in the network virtualization software running in production that was causing connections between our VMs to fail intermittently. The test cluster had a patch for this problem. Although fewer customers were present on the test cluster, we created contention there by occupying almost all the virtual machines in one location – 106 physical hosts, on which we placed 7 or 8 VMs each – and using multiple VMs from each physical host. We chose our distribution of VMs per host to match that observed in the production cluster. In summary, although our results are from a test cluster, they simulate the level of heterogeneity seen in production while letting us operate in a more controlled environment. The EC2 results are also consistent with those from our local testbed. Finally, when we performed

Environment	Scale (VMs)	Experiments
EC2 production	871	Measuring heterogeneity
EC2 test cluster	100-243	Scheduler performance
Local testbed	15	Measuring heterogeneity, scheduler performance
EC2 production	40	Sensitivity analysis

Table 1: Environments used in evaluation.

Load Level	VMs	Write Perf (MB/s)	Std Dev
1 VMs/host	202	61.8	4.9
2 VMs/host	264	56.5	10.0
3 VMs/host	201	53.6	11.2
4 VMs/host	140	46.4	11.9
5 VMs/host	45	34.2	7.9
6 VMs/host	12	25.4	2.5
7 VMs/host	7	24.8	0.9

Table 2: **EC2 Disk Performance vs. VM co-location:** Write performance vs. number of VMs per physical host on EC2. Second column shows how many VMs fell into each load level.

the sensitivity analysis, the problem in the production cluster had been fixed, so we were placed back in the production cluster. We used a controlled sleep workload to achieve reproducible sensitivity experiments, as described in Section 5.4. Table 1 summarizes the environments we used throughout our evaluation.

Our EC2 experiments ran on "small"-size EC2 VMs with 1.7 GB of memory, 1 virtual core with "the equivalent of a 1.0-1.2 GHz 2007 Opteron or Xeon processor," and 160 GB of disk space on potentially shared hard drive [12]. EC2 uses Xen [13] virtualization software.

In all tests, we configured the Hadoop Distributed File System to maintain two replicas of each chunk, and we configured each machine to run up to 2 mappers and 2 reducers simultaneously (the Hadoop default). We chose the data input sizes for our jobs so that each job would run approximately 5 minutes, simulating the shorter, more interactive job-types common in MapReduce [1].

For our workload, we used primarily the Sort benchmark in the Hadoop distribution, but we also evaluated two other MapReduce jobs. Sorting is the main benchmark used for evaluating Hadoop at Yahoo [14], and was also used in Google's paper [1]. In addition, a number of features of sorting make it a desirable benchmark [16].

5.1 Measuring Heterogenity on EC2

Virtualization technology can isolate CPU and memory performance effectively between VMs. However, as explained in Section 3.1, heterogeneity can still arise because I/O devices (disk and network) are shared between VMs. On EC2, VMs get the full available bandwidth when there is no contention, but are reduced to fair sharing when there is contention [12]. We measured the effect of contention on raw disk I/O performance as well as application performance in Hadoop. We saw a difference of 2.5-2.7x between loaded and unloaded machines.

We note that our examples of the effect of load are in some sense extreme, because for small allocations, EC2 seems to try to place a user's virtual machines on different physical hosts. When we allocated 200 or fewer virtual machines, they were all placed on different physical hosts. Our results are also inapplicable to CPU and memory-bound workloads. However, the results are relevant to users running Hadoop at large scales on EC2, because these users will likely have co-located VMs (as we did) and Hadoop is an I/O-intensive workload.

5.1.1 Impact of Contention on I/O Performance

In the first test, we timed a `dd` command that wrote 5000 MB of zeroes from `/dev/zero` to a file in parallel on 871 virtual machines in EC2's production cluster. Because EC2 machines exhibit a "cold start" phenomenon where the first write to a block is slower than subsequent writes, possibly to expand the VM's disk allocation, we "warmed up" 5000 MB of space on each machine before we ran our tests, by running `dd` and deleting its output.

We used a `traceroute` from each VM to an external URL to figure out which physical machine the VM was on – the first hop from a Xen virtual machine is always the `dom0` or supervisor process for that physical host. Our 871 VMs ranged from 202 that were alone on their physical host up to 7 VMs located on one physical host. Table 2 shows average performance and standard deviations. Performance ranged from 62 MB/s for the isolated VMs to 25 MB/s when seven VMs shared a host.

To validate that the performance was tied to contention for disk resources due to multiple VMs writing on the same host, we also tried performing `dd`'s in a smaller EC2 allocation where 200 VMs were assigned to 200 distinct physical hosts. In this environment, `dd` performance was between 51 and 72 MB/s for all but three VMs. These achieved 44, 36 and 17 MB/s respectively. We do not know the cause of these stragglers. The nodes with 44 and 36 MB/s could be explained by contention with other users' VMs given our previous measurements, but the node with 17 MB/s might be a truly faulty machine. From these results, we conclude that background load is an important factor in I/O performance on EC2, and can reduce I/O performance by a factor of 2.5. We also see that stragglers can occur "in the wild" on EC2.

We also measured I/O performance on "large" and

"extra-large" EC2 VMs. These VMs have 2 and 4 virtual disks respectively, which appear to be independent. They achieve 50-60 MB/s performance on each disk. However, a large VM costs 4x more than a small one, and an extra-large costs 8x more. Thus the I/O performance per dollar is on average less than that of small VMs.

5.1.2 Impact of Contention at the Application Level

We also evaluated the hypothesis that background load reduces the performance of Hadoop. For this purpose, we ran two tests with 100 virtual machines: one where each VM was on a separate physical host that was doing no other work, and one where all 100 VMs were packed onto 13 physical hosts, with 7 machines per host. These tests were in EC2's test cluster, where we had allocated all 800 VMs. With both sets of machines, we sorted 100 GB of random data using Hadoop's Sort benchmark with speculative execution disabled (this setting achieved the best performance). With isolated VMs, the job completed in 408s, whereas with VMs packed densely onto physical hosts, it took 1094s. Therefore there is a 2.7x difference in Hadoop performance with a cluster of isolated VMs versus a cluster of colocated VMs.

5.2 Scheduling Experiments on EC2

We evaluated LATE, Hadoop's native scheduler, and no speculation in a variety of experiments on EC2, on clusters of about 200 VMs. For each experiment in this section, we performed 5-7 runs. Due to the environment's variability, some of the results had high variance. To address this issue, we show the average, worst and best-case performance for LATE in our results. We also ran experiments on a smaller local cluster where we had full control over the environment for further validation.

We compared the three schedulers in two settings: Heterogeneous but non-faulty nodes, chosen by assigning a varying number of VMs to each physical host, and an environment with stragglers, created by running CPU and I/O intensive processes on some machines. We wanted to show that LATE provides gains in heterogeneous environments even if there are no faulty nodes.

As described at the start of Section 5, we ran these experiments in an EC2 test cluster where we allocated 800 VMs on 106 physical nodes – nearly the full capacity, since each physical machine seems to support at most 8 VMs – and we selected a subset of the VMs for each test to control colocation and hence contention.

5.2.1 Scheduling in a Heterogeneous Cluster

For our first experiment, we created a heterogeneous cluster by assigning different numbers of VMs to physical hosts. We used 1 to 7 VMs per host, for a total of 243 VMs, as shown in Table 3. We chose this mix to resemble the allocation we saw for 900 nodes in the production EC2 cluster in Section 5.1.

Load Level	Hosts	VMs
1 VMs/host	40	40
2 VMs/host	20	40
3 VMs/host	15	45
4 VMs/host	10	40
5 VMs/host	8	40
6 VMs/host	4	24
7 VMs/host	2	14
Total	99	243

Table 3: Load level mix in our heterogeneous EC2 cluster.

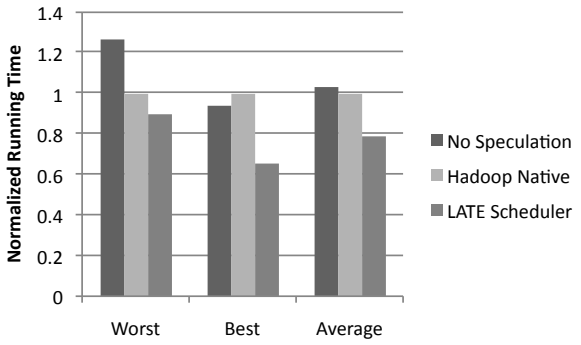


Figure 3: **EC2 Sort running times in heterogeneous cluster:** Worst, best and average-case performance of LATE against Hadoop's scheduler and no speculation.

As our workload, we used a Sort job on a data set of 128 MB per host, or 30 GB of total data. Each job had 486 map tasks and 437 reduce tasks (Hadoop leaves some reduce capacity free for speculative and failed tasks). We repeated the experiment 6 times.

Figure 3 shows the response time achieved by each scheduler. Our graphs throughout this section show normalized performance against that of Hadoop's native scheduler. We show the worst-case and best-case gain from LATE to give an idea of the range involved, because the variance is high. On average, in this first experiment, LATE finished jobs 27% faster than Hadoop's native scheduler and 31% faster than no speculation.

5.2.2 Scheduling with Stragglers

To evaluate the speculative execution algorithms on the problem they were meant to address – faulty nodes – we manually slowed down eight VMs in a cluster of 100 with background processes to simulate stragglers. The other machines were assigned between 1 and 8 VMs per host, with about 10 in each load level. The stragglers

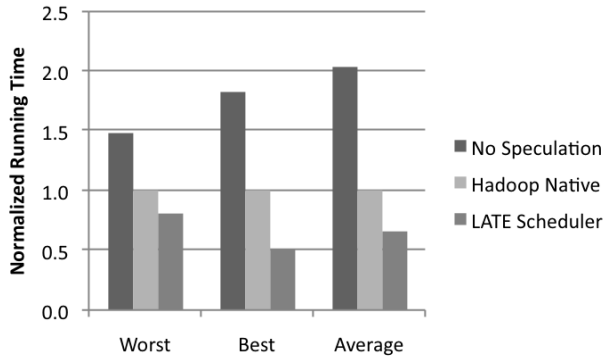


Figure 4: **EC2 Sort running times with stragglers:** Worst, best and average-case performance of LATE against Hadoop's scheduler and no speculation.

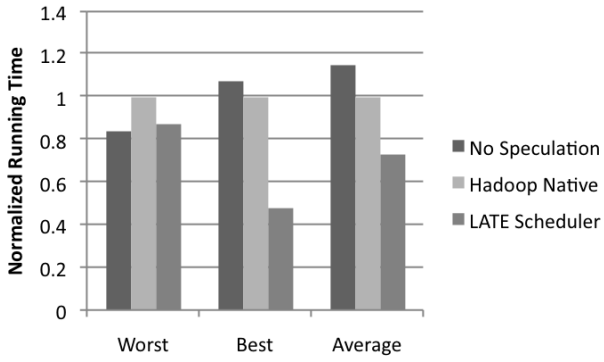


Figure 5: **EC2 Grep running times with stragglers:** Worst, best and average-case performance of LATE against Hadoop's scheduler and no speculation.

were created by running four CPU-intensive processes (tight loops modifying 800 KB arrays) and four disk-intensive processes (`dd` tasks creating large files in a loop) on each straggler. The load was significant enough that disabling speculative tasks caused the cluster to perform 2 to 4 times slower than it did with LATE, but not so significant as to render the straggler machines completely unusable. For each run, we sorted 256 MB of data per host, for a total of 25 GB.

Figure 4 shows the results of 4 experiments. On average, LATE finished jobs 58% faster than Hadoop's native scheduler and 220% faster than Hadoop with speculative execution disabled. The speed improvement over native speculative execution could be as high as 93%.

5.2.3 Differences Across Workloads

To validate our use of the Sort benchmark, we also ran two other workloads, Grep and WordCount, on a heterogeneous cluster with stragglers. These are example jobs that come with the Hadoop distribution. We used a 204-node cluster with 1 to 8 VMs per physical host. We simulated eight stragglers with background load as above.

Grep searches for a regular expression in a text file and creates a file with matches. It then launches a second MapReduce job to sort the matches. We only measured performance of the search job because the sort job was too short for speculative execution to activate (less than a minute). We applied Grep to 43 GB of text data (repeated copies of Shakespeare's plays), or about 200 MB per host. We searched for the regular expression "the". Results from 5 runs are shown in Figure 5. On average, LATE finished jobs 36% faster than Hadoop's native scheduler and 57% faster than no speculation.

We notice that in one of the experiments, LATE performed worse than no speculation. This is not surprising given the variance in the results. We also note that there is an element of "luck" involved in these tests: if a data chunk's two replicas both happen to be placed on stragglers, then no scheduling algorithm can perform very well, because this chunk will be slow to serve.

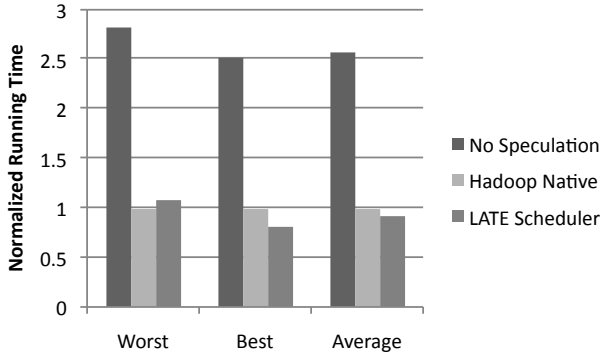


Figure 6: **EC2 WordCount running times with stragglers:** Worst, best and average-case performance of LATE against Hadoop's scheduler and no speculation.

WordCount counts the number of occurrences of each word in a file. We applied WordCount to a smaller data set of 21 GB, or 100 MB per host. Results from 5 runs are shown in Figure 6. On average, LATE finished jobs 8.5% faster than Hadoop's native scheduler and 179% faster than no speculation. We observe that the gain from LATE is smaller in WordCount than in Grep and Sort. This is explained by looking at the workload. Sort and Grep write a significant amount of data over the network and to disk. On the other hand, WordCount only sends a small number of bytes to each reducer – a count for each word. Once the maps in WordCount finish, the reducers finish quickly, so its performance is bound by the mappers. The slowest mappers will be those which read data whose only replicas are on straggler nodes, and therefore

Load Level	VMs	Write Perf (MB/s)	Std Dev
1 VMs/host	5	52.1	13.7
2 VMs/host	6	20.9	2.7
4 VMs/host	4	10.1	1.1

Table 4: **Local cluster disk performance:** Write performance vs. VMs per host on local cluster. The second column shows how many VMs fell into each load level.

Load Level	Hosts	VMs
1 VMs/host	5	5
2 VMs/host	3	6
4 VMs/host	1	4
Total	9	15

Table 5: Load level mix in our heterogeneous local cluster.

they will be equally slow with LATE and native speculation. In contrast, in jobs where reducers do more work, maps are a smaller fraction of the total time, and LATE has more opportunity to outperform Hadoop's scheduler. Nonetheless, speculation was helpful in all tests.

5.3 Local Testbed Experiments

In order to validate our results from EC2 in a more tightly controlled environment, we also ran a local cluster of 9 physical hosts running Xen virtualization software [13].

Our machines were dual-processor, dual-core 2.2 GHz Opteron processors with 4 GB of memory and a single 250GB SATA drive. On each physical machine, we ran one to four virtual machines using Xen, giving each virtual machine 768 MB of memory. While this environment is different from EC2, this appeared to be the most natural way of splitting up the computing resources to allow a large range of virtual machines per host (1-4).

5.3.1 Local I/O Performance Heterogeneity

We first performed a local version of the experiment described in 5.1.1. We started a `dd` command in parallel on each virtual machine which wrote 1GB of zeroes to a file. We captured the timing of each `dd` command and show the averaged results of 10 runs in Table 4. We saw that average write performance ranged from 52.1 MB/s for the isolated VMs to 10.1 MB/s for the 4 VMs that shared a single physical host. We witnessed worse disk I/O performance in our local cluster than on EC2 for the co-located virtual machines because our local nodes each have only a single hard disk, whereas in the worst case on EC2, 8 VMs were contending for 4 disks.

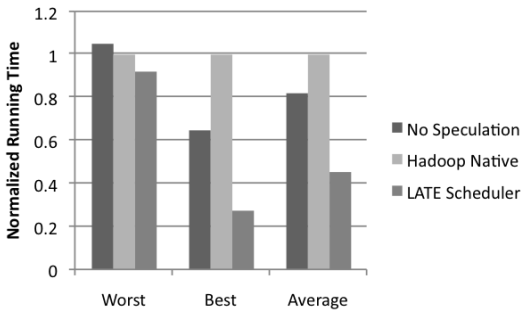


Figure 7: **Local Sort with heterogeneity:** Worst, best and average-case times for LATE against Hadoop's scheduler and no speculation.

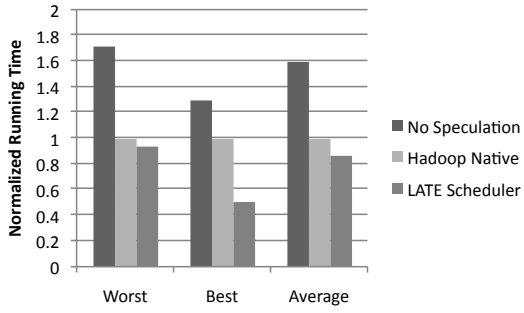


Figure 8: **Local Sort with stragglers:** Worst, best and average-case times for LATE against Hadoop's scheduler and no speculation.

5.3.2 Local Scheduling Experiments

We next configured the local cluster in a heterogeneous fashion to mimic a VM-to-physical-host mapping one might see in a virtualized environment such as EC2. We scaled the allocation to the size of the hardware we were using, as shown in Table 5. We then ran the Hadoop Sort benchmark on 64 MB of input data per node, for 5 runs. Figure 7 shows the results. On average, LATE finished jobs 162% faster than Hadoop's native scheduler and 104% faster than no speculation. The gain over native speculation could be as high as 261%.

We also tested an environment with stragglers by running intensive background processes on two nodes. Figure 8 shows the results. On average, LATE finished jobs 53% faster than Hadoop's native scheduler and 121% faster than Hadoop with speculative execution disabled.

Finally, we also tested the WordCount workload in the local environment with stragglers. The results are shown in Figure 9. We see that LATE performs better on average than the competition, although as on EC2, the gain is less due to the nature of the workload.

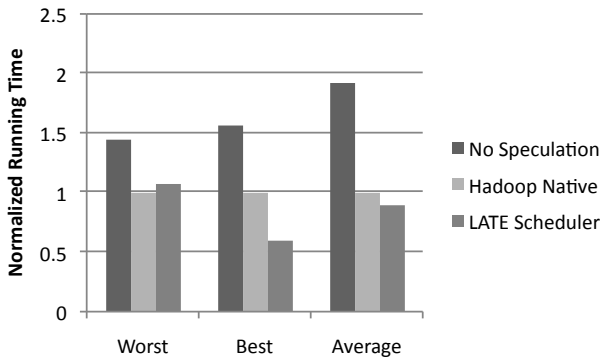


Figure 9: **Local WordCount with stragglers:** Worst, best and average-case times for LATE against Hadoop's scheduler and no speculation.

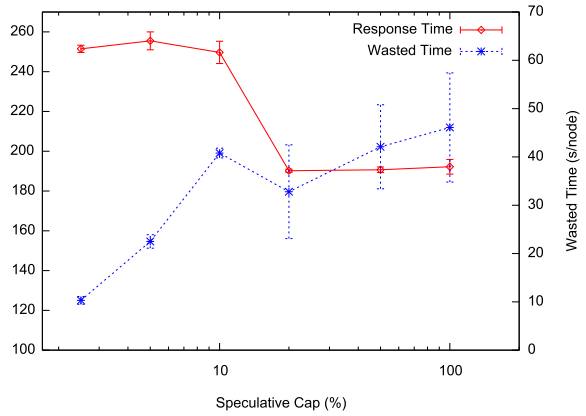


Figure 10: Performance versus SpeculativeCap.

5.4 Sensitivity Analysis

To verify that LATE is not overly sensitive to the thresholds defined in 4, we ran a sensitivity analysis comparing performance at different values for the thresholds. For this analysis, we chose to use a synthetic sleep workload, where each machine was deterministically "slowed down" by a different amount. The reason was that, by this point, we had been moved out of the test cluster in EC2 to the production environment, because the bug that initially put is in the test cluster was fixed. In the production environment, it was more difficult to slow down machines with background load, because we could not easily control VM colocation as we did in the test cluster. Furthermore, there was a risk of other users' traffic causing unpredictable load. To reduce variance and ensure that the experiment is reproducible, we chose to run a synthetic workload based on behavior observed in Sort.

Our job consisted of a fast 15-second map phase followed by a slower reduce phase, motivated by the fact that maps are much faster than reduces in the Sort job. Maps in Sort only read data and bucket it, while reduces merge and sort the output of multiple maps. Our job's reducers chose a sleep time t on each machine based on a per-machine "slowdown factor". They then slept 100 times for random periods between 0 and $2t$, leading to uneven but steady progress. The base sleep time was 0.7 seconds, for a total of 70s per reduce. We ran on 40 machines. The slowdown factors on most machines were 1 or 1.5 (to simulate small variations), but five machines had a sleep factor of 3, and one had a sleep factor of 10, simulating a faulty node.

One flaw in our sensitivity experiments is that the sleep workload does not penalize the scheduler for launching too many speculative tasks, because sleep tasks do not compete for disk and network bandwidth. Nonetheless, we chose this job to make results reproducible. To check that we are not launching too many speculative tasks, we measured the time spent in killed tasks in each test. We also compared LATE to Hadoop's native scheduler and saw that LATE wasted less time speculating while achieving faster response times.

5.4.1 Sensitivity to SpeculativeCap

We started by varying $SpeculativeCap$, that is, the percentage of slots that can be used for speculative tasks at any given time. We kept the other two thresholds, $SlowTaskThreshold$ and $SlowNodeThreshold$, at 25%, which was a well-performing value, as we shall see later. We ran experiments at six $SpeculativeCap$ values from 2.5% to 100%, repeating each one 5 times. Figure 10 shows the results, with error bars for standard deviations. We plot two metrics on this figure: the response time, and the amount of *wasted time* per node, which we define as the total compute time spent in tasks that will eventually be killed (either because they are overtaken by a speculative task, or because an original task finishes before its speculative copy).

We see that response time drops sharply at $SpeculativeCap = 20\%$, after which it stays low. Thus we postulate that any value of $SpeculativeCap$ beyond some minimum threshold needed to speculatively re-execute the severe stragglers will be adequate, as LATE will prioritize the slowest stragglers first. Of course, a higher threshold value is undesirable because LATE wastes more time on excess speculation. However, we see that the amount of wasted time does not grow rapidly, so there is a wide operating range. It is also interesting to note that at a *low* threshold of 10%, we have more wasted time than at 20%, because while fewer speculative tasks are launched, the job runs longer, so more *time* is wasted in tasks that eventually get killed.

As a sanity check, we also ran Hadoop with native

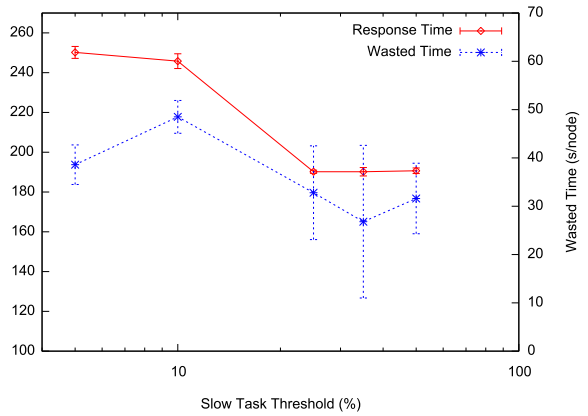


Figure 11: Performance versus SlowTaskThreshold.

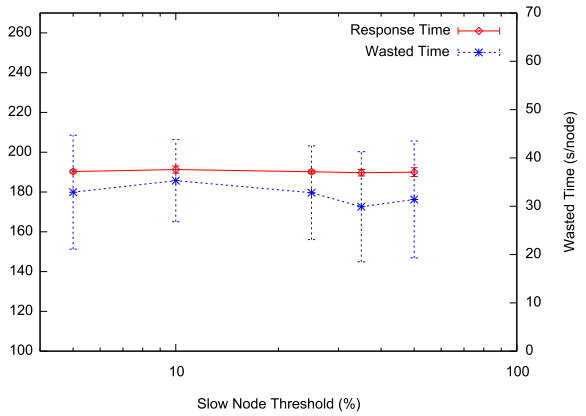


Figure 12: Performance versus SlowNodeThreshold.

speculation and with no speculation. Native speculation had a response time of 247s (std dev 22s), and wasted time of 35s/node (std dev 16s), both of which are worse than LATE with $SlowCapThreshold = 20\%$. No speculation had an average response time of 745s (about 10×70s, as expected) and, of course, 0 wasted time.

Finally, we note that the optimal value for $SpeculativeCap$ in these sensitivity experiments, 20%, was larger than the value we used in our evaluation on EC2, 10%. The 10% threshold probably performed poorly in the sensitivity experiment because 6 out of our 40 nodes, or about 15%, were slow (by 3x or 10x). Unfortunately, it was too late for us to re-run our EC2 test cluster experiments with other values of $SpeculativeCap$, because we no longer had access to the test cluster. Nonetheless, we believe that performance in those experiments could only have gotten better with a larger $SpeculativeCap$, because the sensitivity results presented here show that after some minimum threshold, response time stays low and wasted work does not increase greatly. It is also possible that there were few enough stragglers in the large-scale experiments that a 10% cap was already high enough.

5.4.2 Sensitivity to SlowTaskThreshold

$SlowTaskThreshold$ is the percentile of progress rate below which a task must lie to be considered for speculation (e.g. slowest 5%). The idea is to avoid wasted work by not speculating tasks that are progressing fast when they are the only non-speculated tasks left. For our tests varying this threshold, we set $SpeculativeCap$ to the best value from the previous experiment, 20%, and set $SlowNodeThreshold$ to 25%, a well-performing value. We tested 6 values of $SlowTaskThreshold$, from 5% to 100%. Figure 11 shows the results. We see again that while small threshold values harmfully limit the number of speculative tasks, values past 25% all work well.

5.4.3 Sensitivity to SlowNodeThreshold

$SlowNodeThreshold$ is the percentile of speed below which a node will be considered too slow for LATE to launch speculative tasks on. This value is important because it protects LATE against launching a speculative task on a node that is slow but happens to have a free slot when LATE needs to make a scheduling decision. Launching such a task is wasteful and also means that we cannot re-speculate that task on any other node, because we allow only one speculative copy of each task to run at any time. Unfortunately, our sleep workload did not present any cases in which $SlowNodeThreshold$ was necessary for good performance. This happened because the slowest nodes (3x and 10x slower than the rest) were so slow that they never finished their tasks by the time the job completed, so they were never considered for running speculative tasks.

Nonetheless, it is easy to construct scenarios where $SlowNodeThreshold$ makes a difference. For example, suppose we had 10 fast nodes that could run tasks in 1 minute, one node X that takes 2.9 minutes to run a task, and one node Y that takes 10 minutes to ran a task. Supposed we launched a job with 32 tasks. During the first two minutes, each of the fast nodes would run 2 tasks and be assigned a third. Then, at time 2.9, node X would finish, and there would be no non-speculative task to give it, so it would be assigned a speculative copy of the task on node Y. The job as a whole would finish at time 5.8, when X finishes this speculative task. In contrast, if we waited 0.1 more minutes and assigned the speculative task to a fast node, we would finish at time 4, which is 45% faster. This is why we included $SlowNodeThreshold$ in our algorithm. As long as the threshold is high enough that the very slow nodes fall below it, LATE will make rea-

sonable decisions. Therefore we ran our evaluation with a value of 25%, expecting that fewer than 25% of nodes in a realistic environment will be severe stragglers.

For completeness, Figure 12 shows the results of varying $SlowNodeThreshold$ from 5% to 50% while fixing $SpeculativeCap = 20\%$ and $SlowTaskThreshold = 25\%$. As noted, the threshold has no significant effect on performance. However, it is comforting to see that the very high threshold of 50% did not lead to a *decrease* in performance by unnecessarily limiting the set of nodes we can run speculative tasks on. This further supports the argument that, as long as $SlowNodeThreshold$ is higher than the fraction of nodes that are extremely slow or faulty, LATE performs well.

6 Discussion

Our work is motivated by two trends: increased interest from a variety of organizations in large-scale data-intensive computing, spurred by decreased storage costs and availability of open-source frameworks like Hadoop, and the advent of virtualized data centers, exemplified by Amazon's Elastic Compute Cloud. We believe that both trends will continue. Now that MapReduce has become a well-known technique, more and more organizations are starting to use it. For example, Facebook, a relatively young web company, has built a 300-node data warehouse using Hadoop in the past two years [14]. Many other companies and research groups are also using Hadoop [5, 6]. EC2 also growing in popularity. It powers a number of startup companies, and it has enabled established organizations to rent capacity by the hour for running large computations [17]. Utility computing is also attractive to researchers, because it enables them to run scalability experiments without having to own large numbers of machines, as we did in our paper. Services like EC2 also level the playing field between research institutions by reducing infrastructure costs. Finally, even without utility computing motivating our work, heterogeneity will be a problem in private data centers as multiple generations of hardware accumulate and virtualization starts being used for management and consolidation. These factors mean that dealing with stragglers in MapReduce-like workloads will be an increasingly important problem.

Although selecting speculative tasks initially seems like a simple problem, we have shown that it is surprisingly subtle. First, simple thresholds, such as Hadoop's 20% progress rule, can fail in spectacular ways (see Section 3.2) when there is more heterogeneity than expected. Other work on identifying slow tasks, such as [15], suggests using the mean and the variance of the progress rate to set a threshold, which seems like a more reasonable approach. However, even here there is a problem: identifying *slow* tasks *eventually* is not enough. What matters is identifying *the tasks that will hurt response time the most*, and doing so *as early as possible*. Identifying a task as a laggard when it has run for more than two standard deviations than the mean is not very helpful for reducing response time: by this time, the job could have already run 3x longer than it should have! For this reason, LATE is based on estimated time left, and can detect the slow task early on. A few other elements, such as a cap on speculative tasks, ensure reasonable behavior. Through our experience with Hadoop, we have gained substantial insight into the implications of heterogeneity on distributed applications. We take away four lessons:

1. **Make decisions early,** rather than waiting to base decisions on measurements of mean and variance.
2. **Use finishing times,** not progress rates, to prioritize among tasks to speculate.
3. **Nodes are not equal.** Avoid assigning speculative tasks to slow nodes.
4. **Resources are precious.** Caps should be used to guard against overloading the system.

7 Related Work

MapReduce was described architecturally and evaluated for end-to-end performance in [1]. However, [1] only briefly discusses speculative execution and does not explore the algorithms involved in speculative execution nor the implications of highly variable node performance. Our work provides a detailed look at the problem of speculative execution, motivated by the challenges we observed in heterogeneous environments.

Much work has been done on the problem of scheduling policies for task assignment to hosts in distributed systems [18, 19]. However, this previous work deals with scheduling independent tasks among a set of servers, such as web servers answering HTTP requests. The goal is to achieve good response time for the average *task*, and the challenge is that task sizes may be heterogeneous. In contrast, our work deals with improving response time for a job consisting of *multiple* tasks, and our challenge is that node speeds may be heterogeneous.

Our work is also related to multiprocessor task scheduling with processor heterogeneity [20] and with task duplication when using dependency graphs [21]. Our work differs significantly from this literature because we focus on an environment where node speeds are unknown and vary over time, and where tasks are shared-nothing. Multiprocessor task scheduling work focuses on environments where processor speeds, although heterogeneous, are known in advance, and tasks are highly interdependent due to intertask communication. This

means that, in the multiprocessor setting, it is both possible and necessary to plan task assignments in advance, whereas in MapReduce, the scheduler must react dynamically to conditions in the environment.

Speculative execution in MapReduce shares some ideas with "speculative execution" in distributed file systems [11], configuration management [22], and information gathering [23]. However, while this literature is focused on guessing along decision branches, LATE focuses on guessing which running tasks can be overtaken to reduce the response time of a distributed computation.

Finally, DataSynapse, Inc. holds a patent which details speculative execution for scheduling in a distributed computing platform [15]. The patent proposes using mean speed, normalized mean, standard deviation, and fraction of waiting versus pending tasks associated with each active job to detect slow tasks. However, as discussed in Section 6, detecting slow tasks *eventually* is not sufficient for a good response time. LATE identifies the tasks that will hurt response time the most, and does so as early as possible, rather than waiting until a mean and standard deviation can be computed with confidence.

8 Conclusion

Motivated by the real-world problem of node heterogeneity, we have analyzed the problem of speculative execution in MapReduce. We identified flaws with both the particular threshold-based scheduling algorithm in Hadoop and with progress-rate-based algorithms in general. We designed a simple, robust scheduling algorithm, LATE, which uses estimated finish times to speculatively execute the tasks that hurt the response time the most. LATE performs significantly better than Hadoop's default speculative execution algorithm in real workloads on Amazon's Elastic Compute Cloud.

9 Acknowledgments

We would like to thank Michael Armbrust, Rodrigo Fonseca, George Porter, David Patterson, Armando Fox, and our industrial contacts at Yahoo!, Facebook and Amazon for their help with this work. We are also grateful to our shepherd, Marvin Theimer, for his feedback. This research was supported by California MICRO, California Discovery, the Natural Sciences and Engineering Research Council of Canada, as well as the following Berkeley Reliable Adaptive Distributed systems lab sponsors: Sun, Google, Microsoft, HP, Cisco, Oracle, IBM, NetApp, Fujitsu, VMWare, Siemens, Amazon and Facebook.

References

[1] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In Communications of the ACM, 51 (1): 107-113, 2008.

[2] Hadoop, `http://lucene.apache.org/hadoop`

[3] Amazon Elastic Compute Cloud, `http://aws.amazon.com/ec2`

[4] Yahoo! Launches World's Largest Hadoop Production Application, `http://tinyurl.com/2hgzv7`

[5] Applications powered by Hadoop: `http://wiki.apache.org/hadoop/PoweredBy`

[6] Presentations by S. Schlosser and J. Lin at the 2008 Hadoop Summit. `tinyurl.com/4a6lza`

[7] D. Gottfrid, Self-service, Prorated Super Computing Fun, New York Times Blog, `tinyurl.com/2pjh5n`

[8] Figure from slide deck on MapReduce from Google academic cluster, `tinyurl.com/4zl6f5`. Available under Creative Commons Attribution 2.5 License.

[9] R. Pike, S. Dorward, R. Griesemer, S. Quinlan. Interpreting the Data: Parallel Analysis with Sawzall, Scientific Programming Journal, 13 (4): 227-298, Oct. 2005.

[10] C. Olston, B. Reed, U. Srivastava, R. Kumar and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. ACM SIGMOD 2008, June 2008.

[11] E.B. Nightingale, P.M. Chen, and J.Flinn. Speculative execution in a distributed file system. ACM Trans. Comput. Syst., 24 (4): 361-392, November 2006.

[12] Amazon EC2 Instance Types, `tinyurl.com/3zjlrd`

[13] B.Dragovic, K.Fraser, S.Hand, T.Harris, A.Ho, I.Pratt, A.Warfield, P.Barham, and R.Neugebauer. Xen and the art of virtualization. ACM SOSP 2003.

[14] Personal communication with the Yahoo! Hadoop team and with Joydeep Sen Sarma from Facebook.

[15] J. Bernardin, P. Lee, J. Lewis, DataSynapse, Inc. Using Execution statistics to select tasks for redundant assignment in a distributed computing platform. Patent number 7093004, filed Nov 27, 2002, issued Aug 15, 2006.

[16] G. E. Blelloch, L. Dagum, S. J. Smith, K. Thearling, M. Zagha. An evaluation of sorting as a supercomputer benchmark. NASA Technical Reports, Jan 1993.

[17] EC2 Case Studies, `tinyurl.com/46vyut`

[18] Mor Harchol-Balter, Task Assignment with Unknown Duration. Journal of the ACM, 49 (2): 260-288, 2002.

[19] M.Crovella, M.Harchol-Balter, and C.D. Murta. Task assignment in a distributed system: Improving performance by unbalancing load. In Measurement and Modeling of Computer Systems, pp. 268-269, 1998.

[20] B.Ucar, C.Aykanat, K.Kaya, and M.Ikinci. Task assignment in heterogeneous computing systems. J. of Parallel and Distributed Computing, 66 (1): 32-46, Jan 2006.

[21] S.Manoharan. Effect of task duplication on the assignment of dependency graphs. Parallel Comput., 27 (3): 257-268, 2001.

[22] Y. Su, M. Attariyan, J. Flinn AutoBash: improving configuration management with operating system causality analysis. ACM SOSP 2007.

[23] G. Barish. Speculative plan execution for information agents. PhD dissertation, University of Southernt California. Dec 2003

Corey: An Operating System for Many Cores

Silas Boyd-Wickizer* Haibo Chen† Rong Chen† Yandong Mao†
Frans Kaashoek* Robert Morris* Aleksey Pesterev* Lex Stein‡ Ming Wu‡
Yuehua Dai° Yang Zhang* Zheng Zhang‡
*MIT †Fudan University ‡Microsoft Research Asia °Xi'an Jiaotong University

ABSTRACT

Multiprocessor application performance can be limited by the operating system when the application uses the operating system frequently and the operating system services use data structures shared and modified by multiple processing cores. If the application does not need the sharing, then the operating system will become an unnecessary bottleneck to the application's performance.

This paper argues that *applications should control sharing*: the kernel should arrange each data structure so that only a single processor need update it, unless directed otherwise by the application. Guided by this design principle, this paper proposes three operating system abstractions (address ranges, kernel cores, and shares) that allow applications to control inter-core sharing and to take advantage of the likely abundance of cores by dedicating cores to specific operating system functions.

Measurements of microbenchmarks on the Corey prototype operating system, which embodies the new abstractions, show how control over sharing can improve performance. Application benchmarks, using MapReduce and a Web server, show that the improvements can be significant for overall performance: MapReduce on Corey performs 25% faster than on Linux when using 16 cores. Hardware event counters confirm that these improvements are due to avoiding operations that are expensive on multicore machines.

1 INTRODUCTION

Cache-coherent shared-memory multiprocessor hardware has become the default in modern PCs as chip manufacturers have adopted multicore architectures. Chips with four cores are common, and trends suggest that chips with tens to hundreds of cores will appear within five years [2]. This paper explores new operating system abstractions that allow applications to avoid bottlenecks in the operating system as the number of cores increases.

Operating system services whose performance scales poorly with the number of cores can dominate application performance. Gough *et al.* show that contention for Linux's scheduling queues can contribute significantly to total application run time on two cores [12]. Veal and Foong show that as a Linux Web server uses more cores directory lookups spend increasing amounts of time contending for spin locks [29]. Section 8.5.1 shows that contention for Linux address space data structures causes the percentage of total time spent in the reduce phase of a MapReduce application to increase from 5% at seven cores to almost 30% at 16 cores.

One source of poorly scaling operating system services is use of data structures modified by multiple cores. Figure 1 illustrates such a scalability problem with a simple microbenchmark. The benchmark creates a number of threads within a process, each thread creates a file descriptor, and then each thread repeatedly duplicates (with `dup`) its file descriptor and closes the result. The graph shows results on a machine with four quad-core AMD Opteron chips running Linux 2.6.25. Figure 1 shows that, as the number of cores increases, the total number of `dup` and `close` operations per unit time *decreases*. The cause is contention over shared data: the table describing the process's open files. With one core there are no cache misses, and the benchmark is fast; with two cores, the cache coherence protocol forces a few cache misses per iteration to exchange the lock and table data. More generally, only one thread at a time can update the shared file descriptor table (which prevents any increase in performance), and the increasing number of threads spinning for the lock gradually increases locking costs. This problem is not specific to Linux, but is due to POSIX semantics, which require that a new file descriptor be visible to all of a process's threads even if only one thread uses it.

Common approaches to increasing scalability include avoiding shared data structures altogether, or designing them to allow concurrent access via fine-grained locking or wait-free primitives. For example, the Linux community has made tremendous progress with these approaches [18].

A different approach exploits the fact that some instances of a given resource type need to be shared, while others do not. If the operating system were aware of an application's sharing requirements, it could choose resource implementations suited to those requirements. A

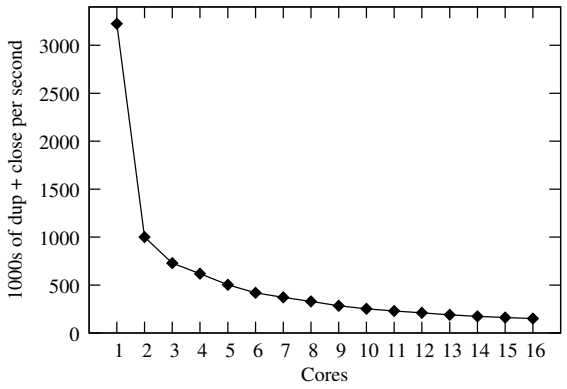


Figure 1: Throughput of the file descriptor `dup` and `close` microbenchmark on Linux.

limiting factor in this approach is that the operating system interface often does not convey the needed information about sharing requirements. In the file descriptor example above, it would be helpful if the thread could indicate whether the new descriptor is private or usable by other threads. In the former case it could be created in a per-thread table without contention.

This paper is guided by a principle that generalizes the above observation: applications should control sharing of operating system data structures. "Application" is meant in a broad sense: the entity that has enough information to make the sharing decision, whether that be an operating system service, an application-level library, or a traditional user-level application. This principle has two implications. First, the operating system should arrange each of its data structures so that by default only one core needs to use it, to avoid forcing unwanted sharing. Second, the operating system interfaces should let callers control how the operating system shares data structures between cores. The intended result is that the operating system incurs sharing costs (e.g., cache misses due to memory coherence) only when the application designer has decided that is the best plan.

This paper introduces three new abstractions for applications to control sharing within the operating system. *Address ranges* allow applications to control which parts of the address space are private per core and which are shared; manipulating private regions involves no contention or inter-core TLB invalidations, while explicit indication of shared regions allows sharing of hardware page tables and consequent minimization of soft page faults (page faults that occur when a core references pages with no mappings in the hardware page table but which are present in physical memory). *Kernel cores* allow applications to dedicate cores to run specific kernel functions, avoiding contention over the data those functions use. *Shares* are lookup tables for kernel objects that allow applications to control which object identifiers are visible to other cores. These abstractions are implementable without inter-core sharing by default, but allow sharing among cores as directed by applications.

We have implemented these abstractions in a prototype operating system called Corey. Corey is organized like an exokernel [9] to ease user-space prototyping of experimental mechanisms. The Corey kernel provides the above three abstractions. Most higher-level services are implemented as library operating systems that use the abstractions to control sharing. Corey runs on machines with AMD Opteron and Intel Xeon processors.

Several benchmarks demonstrate the benefits of the new abstractions. For example, a MapReduce application scales better with address ranges on Corey than on Linux. A benchmark that creates and closes many short TCP connections shows that Corey can saturate the network device with five cores by dedicating a kernel core to manipulating the device driver state, while 11 cores are required when not using a dedicated kernel core. A synthetic Web benchmark shows that Web applications can also benefit from dedicating cores to data.

This paper should be viewed as making a case for controlling sharing rather than demonstrating any absolute conclusions. Corey is an incomplete prototype, and thus it may not be fair to compare it to full-featured operating systems. In addition, changes in the architecture of future multicore processors may change the attractiveness of the ideas presented here.

The rest of the paper is organized as follows. Using architecture-level microbenchmarks, Section 2 measures the cost of sharing on multicore processors. Section 3 describes the three proposed abstractions to control sharing. Section 4 presents the Corey kernel and Section 5 its operating system services. Section 6 summarizes the extensions to the default system services that implement MapReduce and Web server applications efficiently. Section 7 summarizes the implementation of Corey. Section 8 presents performance results. Section 9 reflects on our results so far and outlines directions for future work. Section 10 relates Corey to previous work. Section 11 summarizes our conclusions.

2 MULTICORE CHALLENGES

The main goal of Corey is to allow applications to scale well with the number of cores. This section details some hardware obstacles to achieving that goal.

Future multicore chips are likely to have large total amounts of on-chip cache, but split up among the many cores. Performance may be greatly affected by how well software exploits the caches and the interconnect between cores. For example, using data in a different core's cache is likely to be faster than fetching data from memory, and using data from a nearby core's cache may be faster than using data from a core that is far away on the interconnect.

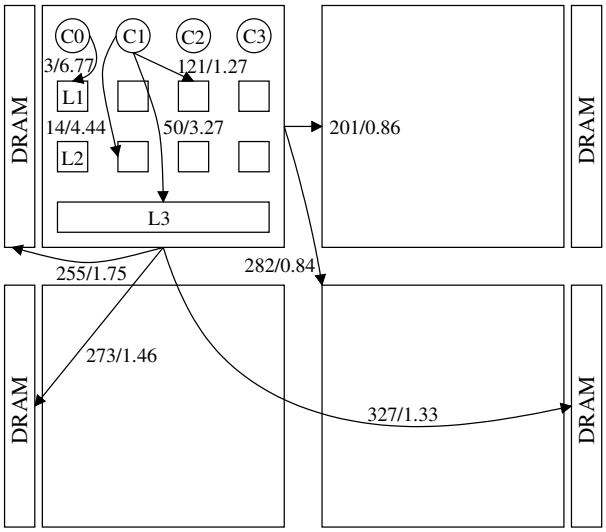


Figure 2: The AMD 16-core system topology. Memory access latency is in cycles and listed before the backslash. Memory bandwidth is in bytes per cycle and listed after the backslash. The measurements reflect the latency and bandwidth achieved by a core issuing load instructions. The measurements for accessing the L1 or L2 caches of a different core on the same chip are the same. The measurements for accessing any cache on a different chip are the same. Each cache line is 64 bytes, L1 caches are 64 Kbytes 8-way set associative, L2 caches are 512 Kbytes 16-way set associative, and L3 caches are 2 Mbytes 32-way set associative.

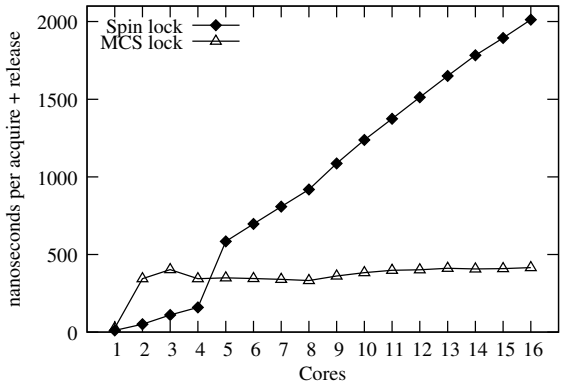


Figure 3: Time required to acquire and release a lock on a 16-core AMD machine when varying number of cores contend for the lock. The two lines show Linux kernel spin locks and MCS locks (on Corey). A spin lock with one core takes about 11 nanoseconds; an MCS lock about 26 nanoseconds.

These properties can already be observed in current multicore machines. Figure 2 summarizes the memory system of a 16-core machine from AMD. The machine has four quad-core Opteron processors connected by a square interconnect. The interconnect carries data between cores and memory, as well as cache coherence broadcasts to locate and invalidate cache lines, and point-to-point cache coherence transfers of individual cache lines. Each core has a private L1 and L2 cache. Four cores on the same chip share an L3 cache. Cores on one chip are connected by an internal crossbar switch and can access each others' L1 and L2 caches efficiently. Memory is partitioned in four banks, each connected to one of the four chips. Each core has a clock frequency of 2.0 GHz.

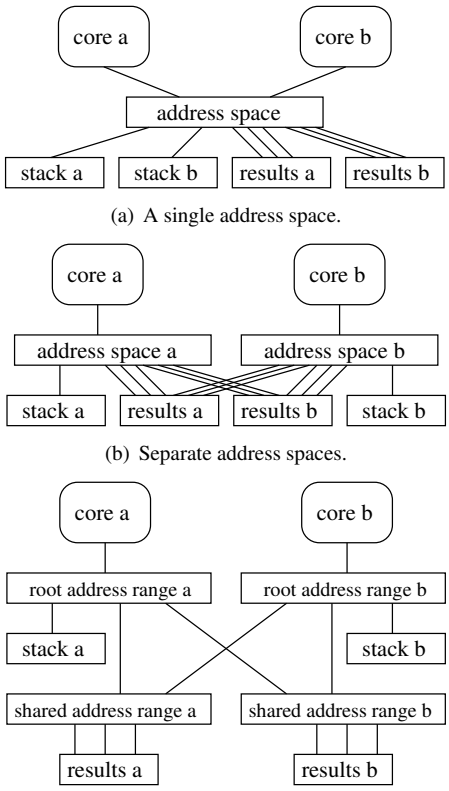
Figure 2 also shows the cost of loading a cache line from a local core, a nearby core, and a distant core. We measured these numbers using many of the techniques documented by Yotov et al. [31]. The techniques avoid interference from the operating system and hardware features such as cache line prefetching.

Reading from the local L3 cache on an AMD chip is faster than reading from the cache of a different core on the same chip. Inter-chip reads are slower, particularly when they go through two interconnect hops.

Figure 3 shows the performance scalability of locking, an important primitive often dominated by cache miss costs. The graph compares Linux's kernel spin locks and scalable MCS locks [21] (on Corey) on the AMD machine, varying the number of cores contending for the lock. For both the kernel spin lock and the MCS lock we measured the time required to acquire and release a single lock. When only one core uses a lock, both types of lock are fast because the lock is always in the core's cache; spin locks are faster than MCS locks because the former requires three instructions to acquire and release when not contended, while the latter requires 15. The time for each successful acquire increases for the spin lock with additional cores because each new core adds a few more cache coherence broadcasts per acquire, in order for the new core to observe the effects of other cores' acquires and releases. MCS locks avoid this cost by having each core spin on a separate location.

We measured the average times required by Linux 2.6.25 to flush the TLBs of all cores after a change to a shared page table. These "TLB shootdowns" are considerably slower on 16 cores than on 2 (18742 cycles versus 5092 cycles). TLB shootdown typically dominates the cost of removing a mapping from an address space that is shared among multiple cores.

On AMD 16-core systems, the speed difference between a fast memory access and a slow memory access is a factor of 100, and that factor is likely to grow with the number of cores. Our measurements for the Intel Xeon show a similar speed difference. For performance, kernels must mainly access data in the local core's cache. A contended or falsely shared cache line decreases performance because many accesses are to remote caches. Widely-shared and contended locks also decrease performance, even if the critical sections they protect are only a few instructions. These performance effects will be more damaging as the number of cores increases, since the cost of accessing far-away caches increases with the number of cores.



(a) A single address space.

(b) Separate address spaces.

(c) Two address spaces with shared result mappings.

Figure 4: Example address space configurations for MapReduce executing on two cores. Lines represent mappings. In this example a stack is one page and results are three pages.

3 DESIGN

Existing operating system abstractions are often difficult to implement without sharing kernel data among cores, regardless of whether the application needs the shared semantics. The resulting unnecessary sharing and resulting contention can limit application scalability. This section gives three examples of unnecessary sharing and for each example introduces a new abstraction that allows the application to decide if and how to share. The intent is that these abstractions will help applications to scale to large numbers of cores.

3.1 Address ranges

Parallel applications typically use memory in a mixture of two sharing patterns: memory that is used on just one core (private), and memory that multiple cores use (shared). Most operating systems give the application a choice between two overall memory configurations: a single address space shared by all cores or a separate address space per core. The term address space here refers to the description of how virtual addresses map to memory, typically defined by kernel data structures and instantiated lazily (in response to soft page faults) in hardware-defined page tables or TLBs. If an application chooses to harness concurrency using threads, the result is typically a single address space shared by all threads. If an application obtains concurrency by forking multiple processes, the result is typically a private address space per process; the processes can then map shared segments into all address spaces. The problem is that each of these two configurations works well for only one of the sharing patterns, placing applications with a mixture of patterns in a bind.

As an example, consider a MapReduce application [8]. During the map phase, each core reads part of the application's input and produces intermediate results; map on each core writes its intermediate results to a different area of memory. Each map instance adds pages to its address space as it generates intermediate results. During the reduce phase each core reads intermediate results produced by multiple map instances to produce the output.

For MapReduce, a single address-space (see Figure 4(a)) and separate per-core address-spaces (see Figure 4(b)) incur different costs. With a single address space, the map phase causes contention as all cores add mappings to the kernel's address space data structures. On the other hand, a single address space is efficient for reduce because once any core inserts a mapping into the underlying hardware page table, all cores can use the mapping without soft page faults. With separate address spaces, the map phase sees no contention while adding mappings to the per-core address spaces. However, the reduce phase will incur a soft page fault per core per page of accessed intermediate results. Neither memory configuration works well for the entire application.

We propose address ranges to give applications high performance for both private and shared memory (see Figure 4(c)). An address range is a kernel-provided abstraction that corresponds to a range of virtual-to-physical mappings. An application can allocate address ranges, insert mappings into them, and place an address range at a desired spot in the address space. If multiple cores' address spaces incorporate the same address range, then they will share the corresponding pieces of hardware page tables, and see mappings inserted by each others' soft page faults. A core can update the mappings in a non-shared address range without contention. Even when shared, the address range is the unit of locking; if only one core manipulates the mappings in a shared address range, there will be no contention. Finally, deletion of mappings from a non-shared address range does not require TLB shootdowns.

Address ranges allow applications to selectively share parts of address spaces, instead of being forced to make an all-or-nothing decision. For example, the MapReduce runtime can set up address spaces as shown in 4(c). Each core has a private root address range that maps all private memory segments used for stacks and temporary objects

and several shared address ranges mapped by all other cores. A core can manipulate mappings in its private address ranges without contention or TLB shootdowns. If each core uses a different shared address range to store the intermediate output of its map phase (as shown Figure 4(c)), the map phases do not contend when adding mappings. During the reduce phase there are no soft page faults when accessing shared segments, since all cores share the corresponding parts of the hardware page tables.

3.2 Kernel cores

In most operating systems, when application code on a core invokes a system call, the same core executes the kernel code for the call. If the system call uses shared kernel data structures, it acquires locks and fetches relevant cache lines from the last core to use the data. The cache line fetches and lock acquisitions are costly if many cores use the same shared kernel data. If the shared data is large, it may end up replicated in many caches, potentially reducing total effective cache space and increasing DRAM references.

We propose a kernel core abstraction that allows applications to dedicate cores to kernel functions and data. A kernel core can manage hardware devices and execute system calls sent from other cores. For example, a Web service application may dedicate a core to interacting with the network device instead of having all cores manipulate and contend for driver and device data structures (e.g., transmit and receive DMA descriptors). Multiple application cores then communicate with the kernel core via shared-memory IPC; the application cores exchange packet buffers with the kernel core, and the kernel core manipulates the network hardware to transmit and receive the packets.

This plan reduces the number of cores available to the Web application, but it may improve overall performance by reducing contention for driver data structures and associated locks. Whether a net performance improvement would result is something that the operating system cannot easily predict. Corey provides the kernel core abstraction so that applications can make the decision.

3.3 Shares

Many kernel operations involve looking up identifiers in tables to yield a pointer to the relevant kernel data structure; file descriptors and process IDs are examples of such identifiers. Use of these tables can be costly when multiple cores contend for locks on the tables and on the table entries themselves.

For each kind of lookup, the operating system interface designer or implementer typically decides the scope of sharing of the corresponding identifiers and tables. For example, Unix file descriptors are shared among the threads of a process. Process identifiers, on the other hand, typically have global significance. Common implementations use per-process and global lookup tables, respectively. If a particular identifier used by an application needs only limited scope, but the operating system implementation uses a more global lookup scheme, the result may be needless contention over the lookup data structures.

We propose a share abstraction that allows applications to dynamically create lookup tables and determine how these tables are shared. Each of an application's cores starts with one share (its root share), which is private to that core. If two cores want to share a share, they create a share and add the share's ID to their private root share (or to a share reachable from their root share). A root share doesn't use a lock because it is private, but a shared share does. An application can decide for each new kernel object (including a new share) which share will hold the identifier.

Inside the kernel, a share maps application-visible identifiers to kernel data pointers. The shares reachable from a core's root share define which identifiers the core can use. Contention may arise when two cores manipulate the same share, but applications can avoid such contention by placing identifiers with limited sharing scope in shares that are only reachable on a subset of the cores.

For example, shares could be used to implement file-descriptor-like references to kernel objects. If only one thread uses a descriptor, it can place the descriptor in its core's private root share. If the descriptor is shared between two threads, these two threads can create a share that holds the descriptor. If the descriptor is shared among all threads of a process, the file descriptor can be put in a per-process share. The advantage of shares is that the application can limit sharing of lookup tables and avoid unnecessary contention if a kernel object is not shared. The downside is that an application must often keep track of the share in which it has placed each identifier.

4 COREY KERNEL

Corey provides a kernel interface organized around five types of low-level objects: shares, segments, address ranges, pcores, and devices. Library operating systems provide higher-level system services that are implemented on top of these five types. Applications implement domain specific optimizations using the low-level interface; for example, using pcores and devices they can implement kernel cores. Figure 5 provides an overview of the Corey low-level interface. The following sections describe the design of the five types of Corey kernel objects and how they allow applications to control sharing. Section 5 describes Corey's higher-level system services.

System call	Description
name obj_get_name(obj)	return the name of an object
shareid share_alloc(shareid, name, memid)	allocate a share object
void share_addobj(shareid, obj)	add a reference to a shared object to the specified share
void share_delobj(obj)	remove an object from a share, decrementing its reference count
void self_drop(shareid)	drop current core's reference to a share
segid segment_alloc(shareid, name, memid)	allocate physical memory and return a segment object for it
segid segment_copy(shareid, seg, name, mode)	copy a segment, optionally with copy-on-write or -read
nbytes segment_get_nbytes(seg)	get the size of a segment
void segment_set_nbytes(seg, nbytes)	set the size of a segment
arid ar_alloc(shareid, name, memid)	allocate an address range object
void ar_set_seg(ar, voff, segid, soff, len)	map addresses at voff in ar to a segment's physical pages
void ar_set_ar(ar, voff, ar1, aoff, len)	map addresses at voff in ar to address range ar1
ar_mappings ar_get(ar)	return the address mappings for a given address range
pcoreid pcore_alloc(shareid, name, memid)	allocate a physical core object
pcore pcore_current(void)	return a reference to the object for current pcore
void pcore_run(pcore, context)	run the specified user context
void pcore_add_device(pcore, dev)	specify device list to a kernel core
void pcore_set_interval(pcore, hz)	set the time slice period
void pcore_halt(pcore)	halt the pcore
devid device_alloc(shareid, hwid, memid)	allocate the specified device and return a device object
dev_list device_list(void)	return the list of devices
dev_stat device_stat(dev)	return information about the device
void device_conf(dev, dev_conf)	configure a device
void device_buf(dev, seg, offset, buf_type)	feed a segment to the device object
locality_matrix locality_get(void)	get hardware locality information

Figure 5: Corey system calls. shareid, segid, arid, pcoreid, and devid represent 64-bit object IDs. share, seg, ar, pcore, obj, and dev represent ⟨share ID, object ID⟩ pairs. hwid represents a unique ID for hardware devices and memid represents a unique ID for per-core free page lists.

4.1 Object metadata

The kernel maintains metadata describing each object. To reduce the cost of allocating memory for object metadata, each core keeps a local free page list. If the architecture is NUMA, a core's free page list holds pages from its local memory node. The system call interface allows the caller to indicate which core's free list a new object's memory should be taken from. Kernels on all cores can address all object metadata since each kernel maps all of physical memory into its address space.

The Corey kernel generally locks an object's metadata before each use. If the application has arranged things so that the object is only used on one core, the lock and use of the metadata will be fast (see Figure 3), assuming they have not been evicted from the core's cache. Corey uses spin lock and read-write lock implementations borrowed from Linux, its own MCS lock implementation, and a scalable read-write lock implementation inspired by the MCS lock to synchronize access to object metadata.

4.2 Object naming

An application allocates a Corey object by calling the corresponding `alloc` system call, which returns a unique 64-bit object ID. In order for a kernel to use an object, it must know the object ID (usually from a system call argument), and it must map the ID to the address of the object's metadata. Corey uses shares for this purpose. Applications specify which shares are available on which cores by passing a core's share set to `pcore_run` (see below).

When allocating an object, an application selects a share to hold the object ID. The application uses ⟨share ID, object ID⟩ pairs to specify objects to system calls. Applications can add a reference to an object in a share with `share_addobj` and remove an object from a share with `share_delobj`. The kernel counts references to each object, freeing the object's memory when the count is zero. By convention, applications maintain a per-core private share and one or more shared shares.

4.3 Memory management

The kernel represents physical memory using the *segment* abstraction. Applications use `segment_alloc` to allocate segments and `segment_copy` to copy a segment or mark the segment as copy-on-reference or copy-on-write. By default, only the core that allocated the segment can reference it; an application can arrange to share a segment between cores by adding it to a share, as described above.

An application uses address ranges to define its address space. Each running core has an associated root address range object containing a table of address mappings. By convention, most applications allocate a root address range for each core to hold core private map-

pings, such as thread stacks, and use one or more address ranges that are shared by all cores to hold shared segment mappings, such as dynamically allocated buffers. An application uses `ar_set_seg` to cause an address range to map addresses to the physical memory in a segment, and `ar_set_ar` to set up a tree of address range mappings.

4.4 Execution

Corey represents physical cores with *pcore* objects. Once allocated, an application can start execution on a physical core by invoking `pcore_run` and specifying a pcore object, instruction and stack pointer, a set of shares, and an address range. A pcore executes until `pcore_halt` is called. This interface allows Corey to space-multiplex applications over cores, dedicating a set of cores to a given application for a long period of time, and letting each application manage its own cores.

An application configures a kernel core by allocating a pcore object, specifying a list of devices with `pcore_add_device`, and invoking `pcore_run` with the kernel core option set in the context argument. A kernel core continuously polls the specified devices by invoking a device specific function. A kernel core polls both real devices and special "syscall" pseudo-devices.

A *syscall device* allows an application to invoke system calls on a kernel core. The application communicates with the *syscall device* via a ring buffer in a shared memory segment.

5 System services

This section describes three system services exported by Corey: execution forking, network access, and a buffer cache. These services together with a C standard library that includes support for file descriptors, dynamic memory allocation, threading, and other common Unix-like features create a scalable and convenient application environment that is used by the applications discussed in Section 6.

5.1 `cfork`

`cfork` is similar to Unix `fork` and is the main abstraction used by applications to extend execution to a new core. `cfork` takes a physical core ID, allocates a new pcore object and runs it. By default, `cfork` shares little state between the parent and child pcore. The caller marks most segments from its root address range as copy-on-write and maps them into the root address range of the new pcore. Callers can instruct `cfork` to share specified segments and address ranges with the new processor. Applications implement fine-grained sharing using shared segment mappings and more coarse-grained sharing using shared address range mappings. `cfork` callers can share kernel objects with the new pcore by passing a set of shares for the new pcore.

5.2 Network

Applications can choose to run several network stacks (possibly one for each core) or a single shared network stack. Corey uses the lwIP [19] networking library. Applications specify a network device for each lwIP stack. If multiple network stacks share a single physical device, Corey virtualizes the network card. Network stacks on different cores that share a physical network device also share the device driver data, such as the transmit descriptor table and receive descriptor table.

All configurations we have experimented with run a separate network stack for each core that requires network access. This design provides good scalability but requires multiple IP addresses per server and must balance requests using an external mechanism. A potential solution is to extend the Corey virtual network driver to use ARP negotiation to balance traffic between virtual network devices and network stacks (similar to the Linux Ethernet bonding driver).

5.3 Buffer cache

An inter-core shared buffer cache is important to system performance and often necessary for correctness when multiple cores access shared files. Since cores share the buffer cache they might contend on the data structures used to organize cached disk blocks. Furthermore, under write-heavy workloads it is possible that cores will contend for the cached disk blocks.

The Corey buffer cache resembles a traditional Unix buffer cache; however, we found three techniques that substantially improve multicore performance. The first is a lock-free tree that allows multiple cores to locate cached blocks without contention. The second is a write scheme that tries to minimize contention on shared data using per-core block allocators and by copying application data into blocks likely to be held in local hardware caches. The third uses a scalable read-write lock to ensure blocks are not freed or reused during reads.

6 Applications

It is unclear how big multicore systems will be used, but parallel computation and network servers are likely application areas. To evaluate Corey in these areas, we implemented a MapReduce system and a Web server with synthetic dynamic content. Both applications are data-parallel but have sharing requirements and thus are not trivially parallelized. This section describes how the two applications use the Corey interface to achieve high performance.

6.1 MapReduce applications

MapReduce is a framework for data-parallel execution of simple programmer-supplied functions. The programmer need not be an expert in parallel programming to

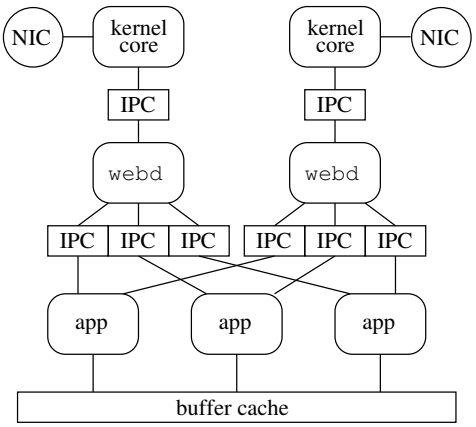


Figure 6: A Corey Web server configuration with two kernel cores, two `webd` cores and three application cores. Rectangles represent segments, rounded rectangles represents pcores, and circles represent devices.

achieve good parallel performance. Data-parallel applications fit well with the architecture of multicore machines, because each core has its own private cache and can efficiently process data in that cache. If the runtime does a good job of putting data in caches close to the cores that manipulate that data, performance should increase with the number of cores. The difficult part is the global communication between the map and reduce phases.

We started with the Phoenix MapReduce implementation [22], which is optimized for shared-memory multiprocessors. We reimplemented Phoenix to simplify its implementation, to use better algorithms for manipulating the intermediate data, and to optimize its performance. We call this reimplementation Metis.

Metis on Corey exploits address ranges as described in Section 3. Metis uses a separate address space on each core, with private mappings for most data (e.g. local variables and the input to map), so that each core can update its own page tables without contention. Metis uses address ranges to share the output of the map on each core with the reduce phase on other cores. This arrangement avoids contention as each map instance adds pages to hold its intermediate output, and ensures that the reduce phase incurs no soft page faults while processing intermediate data from the map phase.

6.2 Web server applications

The main processing in a Web server includes low-level network device handling, TCP processing, HTTP protocol parsing and formatting, application processing (for dynamically generated content), and access to application data. Much of this processing involves operating system services. Different parts of the processing require different parallelization strategies. For example, HTTP parsing is easy to parallelize by connection, while application processing is often best parallelized by partitioning application data across cores to avoid multiple cores contending for the same data. Even for read-only application data, such partitioning may help maximize the amount of distinct data cached by avoiding duplicating the same data in many cores' local caches.

The Corey Web server is built from three components: Web daemons (`webd`), kernel cores, and applications (see Figure 6). The components communicate via shared-memory IPC. `Webd` is responsible for processing HTTP requests. Every core running a `webd` front-end uses a private TCP/IP stack. `Webd` can manipulate the network device directly or use a kernel core to do so. If using a kernel core, the TCP/IP stack of each core passes packets to transmit and buffers to receive incoming packets to the kernel core using a syscall device. `Webd` parses HTTP requests and hands them off to a core running application code. The application core performs the required computation and returns the results to `webd`. `Webd` packages the results in an HTTP response and transmits it, possibly using a kernel core. Applications may run on dedicated cores (as shown in Figure 6) or run on the same core as a `webd` front-end.

All kernel objects necessary for a `webd` core to complete a request, such as packet buffer segments, network devices, and shared segments used for IPC are referenced by a private per-`webd` core share. Furthermore, most kernel objects, with the exception of the IPC segments, are used by only one core. With this design, once IPC segments have been mapped into the `webd` and application address ranges, cores process requests without contending over any global kernel data or locks, except as needed by the application.

The application running with `webd` can run in two modes: *random* mode and *locality* mode. In random mode, `webd` forwards each request to a randomly chosen application core. In locality mode, `webd` forwards each request to a core chosen from a hash of the name of the data that will be needed in the request. Locality mode increases the probability that the application core will have the needed data in its cache, and decreases redundant caching of the same data.

7 IMPLEMENTATION

Corey runs on AMD Opteron and the Intel Xeon processors. Our implementation is simplified by using a 64-bit virtual address space, but nothing in the Corey design relies on a large virtual address space. The implementation of address ranges is geared towards architectures with hardware page tables. Address ranges could be ported to TLB-only architectures, such as MIPS, but would provide less performance benefit, because every core must fill its own TLB.

The Corey implementation is divided into two parts: the low-level code that implements Corey objects (described in Section 4) and the high-level Unix-like environment (described in Section 5). The low-level code (the kernel) executes in the supervisor protection domain. The high-level Unix-like services are a library that applications link with and execute in application protection domains.

The low-level kernel implementation, which includes architecture specific functions and device drivers, is 11,000 lines of C and 150 lines of assembly. The Unix-like environment, 11,000 lines of C/C++, provides the buffer cache, `cfork`, and TCP/IP stack interface as well as the Corey-specific glue for the uClibc [28] C standard library, lwIP, and the Streamflow [24] dynamic memory allocator. We fixed several bugs in Streamflow and added support for x86-64 processors.

8 EVALUATION

This section demonstrates the performance improvements that can be obtained by allowing applications to control sharing. We make these points using several microbenchmarks evaluating address ranges, kernel cores, and shares independently, as well as with the two applications described in Section 6.

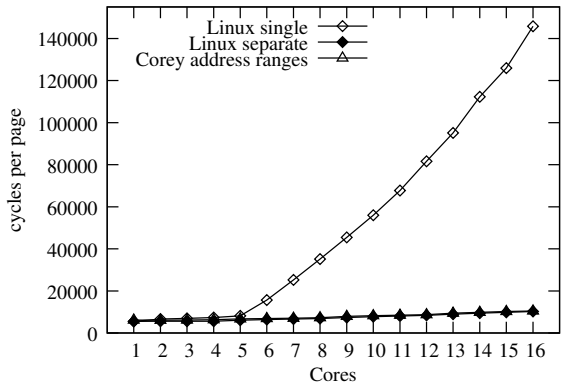
8.1 Experimental setup

We ran all experiments on an AMD 16-core system (see Figure 2 in Section 2) with 64 Gbytes of memory. We counted the number of cache misses and computed the average latency of cache misses using the AMD hardware event counters. All Linux experiments use Debian Linux with kernel version 2.6.25 and pin one thread to each core. The kernel is patched with perfctr 2.6.35 to allow application access to hardware event counters. For Linux MapReduce experiments we used our version of Streamflow because it provided better performance than other allocators we tried, such as TCMalloc [11] and glibc 2.7 malloc. All network experiments were performed on a gigabit switched network, using the server's Intel Pro/1000 Ethernet device.

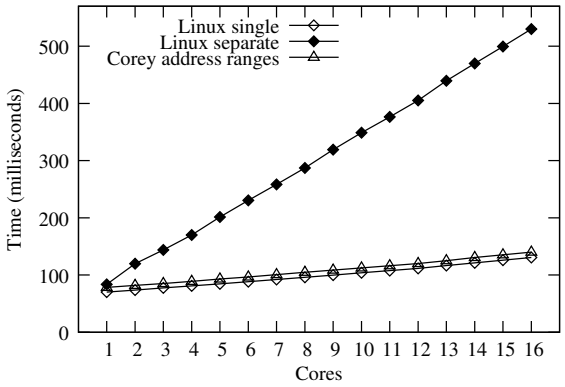
We also have run many of the experiments with Corey and Linux on a 16-core Intel machine and with Windows on 16-core AMD and Intel machines, and we draw conclusions similar to the ones reported in this section.

8.2 Address ranges

To evaluate the benefits of address ranges in Corey, we need to investigate two costs in multicore applications where some memory is private and some is shared. First, the contention costs of manipulating mappings for private memory. Second, the soft page-fault costs for memory that is used on multiple cores. We expect Corey to have low costs for both situations. We expect other systems (represented by Linux in these experiments) to have



(a) `memclone`



(b) `mempass`

Figure 7: Address ranges microbenchmark results.

low cost for only one type of sharing, but not both, depending on whether the application uses a single address space shared by all cores or a separate address space per core.

The `memclone` [3] benchmark explores the costs of private memory. `Memclone` has each core allocate a 100 Mbyte array and modify each page of its array. The memory is demand-zero-fill: the kernel initially allocates no memory, and allocates pages only in response to page faults. The kernel allocates the memory from the DRAM system connected to the core's chip. The benchmark measures the average time to modify each page. `Memclone` allocates cores from chips in a round-robin fashion. For example, when using five cores, `memclone` allocates two cores from one chip and one core from each of the other three chips.

Figure 7(a) presents the results for three situations: Linux with a single address space shared by per-core threads, Linux with a separate address space (process) per core but with the 100 Mbyte arrays `mmap`ed into each process, and Corey with separate address spaces but with the arrays mapped via shared address ranges.

`Memclone` scales well on Corey and on Linux with separate address spaces, but poorly on Linux with a single address space. On a page fault both Corey and Linux

verify that the faulting address is valid, allocate and clear a 4 Kbyte physical page, and insert the physical page into the hardware page table. Clearing a 4 Kbyte page incurs 64 L3 cache misses and copies 4 Kbytes from DRAM to the local L1 cache and writes 4 Kbytes from the L3 back to DRAM; however, the AMD cache line prefetcher allows a core to clear a page in 3350 cycles (or at a rate of 1.2 bytes per cycle).

Corey and Linux with separate address spaces each incur only 64 cache misses per page fault, regardless of the number of cores. However, the cycles per page gradually increases because the per-chip memory controller can only clear 1.7 bytes per cycle and is saturated when `memclone` uses more than two cores on a chip.

For Linux with a single address space cores contend over shared address space data structures. With six cores the cache-coherency messages generated from locking and manipulating the shared address space begin to congest the interconnect, which increases the cycles required for processing a page fault. Processing a page fault takes 14 times longer with 16 cores than with one.

We use a benchmark called `mempass` to evaluate the costs of memory that is used on multiple cores. `Mempass` allocates a single 100 Mbyte shared buffer on one of the cores, touches each page of the buffer, and then passes it to another core, which repeats the process until every core touches every page. `Mempass` measures the total time for every core to touch every page.

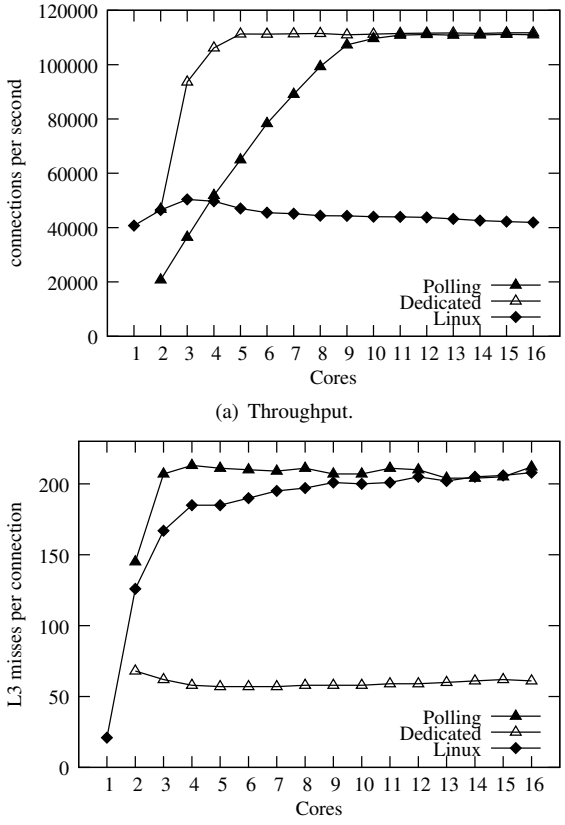
Figure 7(b) presents the results for the same configurations used in `memclone`. This time Linux with a single address space performs well, while Linux with separate address spaces performs poorly. Corey performs well here too. Separate address spaces are costly with this workload because each core separately takes a soft page fault for the shared page.

To summarize, a Corey application can use address ranges to get good performance for both shared and private memory. In contrast, a Linux application can get good performance for only one of these types of memory.

8.3 Kernel cores

This section explores an example in which use of the Corey kernel core abstraction improves scalability. The benchmark application is a simple TCP service, which accepts incoming connections, writing 128 bytes to each connection before closing it. Up to 15 separate client machines (as many as there are active server cores) generate the connection requests.

We compare two server configurations. One of them (called "Dedicated") uses a kernel core to handle all network device processing: placing packet buffer pointers in device DMA descriptors, polling for received packets and transmit completions, triggering device transmissions, and manipulating corresponding driver data structures. The second configuration, called "Polling", uses a kernel core only to poll for received packet notifications and transmit completions. In both cases, each other core runs a private TCP/IP stack and an instance of the TCP service. For Dedicated, each service core uses shared-memory IPC to send and receive packet buffers with the kernel core. For Polling, each service core transmits packets and registers receive packet buffers by directly manipulating the device DMA descriptors (with locking), and is notified of received packets via IPC from the Polling kernel core. The purpose of the comparison is to show the effect on performance of moving all device processing to the Dedicated kernel core, thereby eliminating contention over device driver data structures. Both configurations poll for received packets, since otherwise interrupt costs would dominate performance.



(a) Throughput.

(b) L3 cache misses.

Figure 8: TCP microbenchmark results.

Figure 8(a) presents the results of the TCP benchmark. The network device appears to be capable of handling only about 900,000 packets per second in total, which limits the throughput to about 110,000 connections per second in all configurations (each connection involves 4 input and 4 output packets). The dedicated configuration reaches 110,000 with only five cores, while Polling re-

quires 11 cores. That is, each core is able to handle more connections per second in the Dedicated configuration than in Polling.

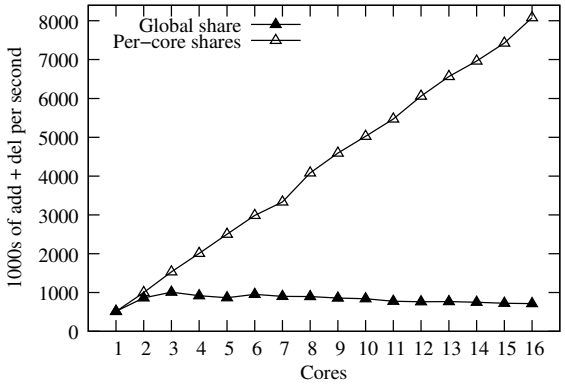
Most of this difference in performance is due to the Dedicated configuration incurring fewer L3 misses. Figure 8(b) shows the number of L3 misses per connection. Dedicated incurs about 50 L3 misses per connection, or slightly more than six per packet. These include misses to manipulate the IPC data structures (shared with the core that sent or received the packet) and the device DMA descriptors (shared with the device) and the misses for a core to read a received packet. Polling incurs about 25 L3 misses per packet; the difference is due to misses on driver data structures (such as indexes into the DMA descriptor rings) and the locks that protect them. To process a packet the device driver must acquire and release an MCS lock twice: once to place the packet on DMA ring buffer and once to remove it. Since each L3 miss takes about 100 nanoseconds to service, misses account for slightly less than half the total cost for the Polling configuration to process a connection (with two cores). This analysis is consistent with Dedicated processing about twice as many connections with two cores, since Dedicated has many fewer misses. This approximate relationship holds up to the 110,000 connection per second limit.

The Linux results are presented for reference. In Linux every core shares the same network stack and a core polls the network device when the packet rate is high. All cores place output packets on a software transmit queue. Packets may be removed from the software queue by any core, but usually the polling core transfers the packets from the software queue to the hardware transmit buffer on the network device. On packet reception, the polling core removes the packet from the ring buffer and performs the processing necessary to place the packet on the queue of a TCP socket where it can be read by the application. With two cores, most cycles on both cores are used by the kernel; however, as the number of cores increases the polling core quickly becomes a bottleneck and other cores have many idle cycles.

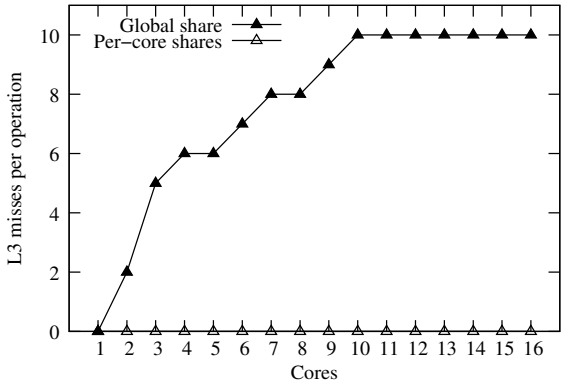
The TCP microbenchmark results demonstrate that a Corey application can use kernel cores to improve scalability by dedicating cores to handling kernel functions and data.

8.4 Shares

We demonstrate the benefits of shares on Corey by comparing the results of two microbenchmarks. In the first microbenchmark each core calls `share_addobj` to add a per-core segment to a global share and then removes the segment from the share with `share_delobj`. The benchmark measures the throughput of add and remove operations. As the number of cores increases, cores contend on the scalable read-write lock protecting the hashtable used to implement the share. The second microbenchmark is similar to the first, except each core adds a per-core segment to a local share so that the cores do not contend.



(a) Throughput.



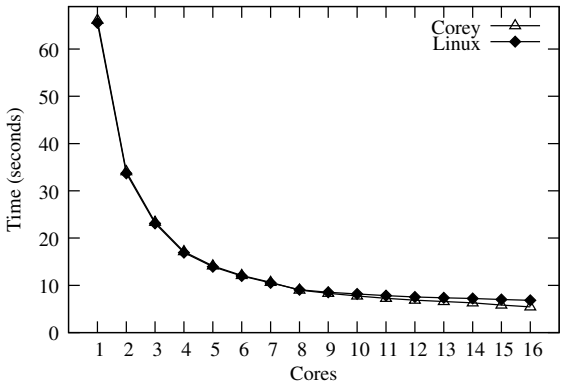
(b) L3 cache misses.

Figure 9: Share microbenchmark results.

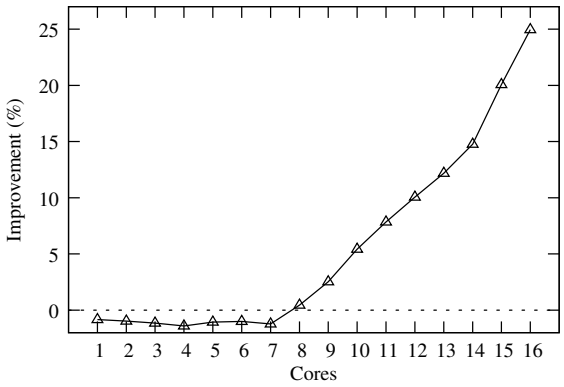
Figure 9 presents the results. The global share scales relatively well for the first three cores, but at four cores performance begins to slowly decline. Cores contend on the global share, which increases the amount of cache coherence traffic and makes manipulating the global share more expensive. The local shares scale perfectly. At 16 cores adding to and removing from the global share costs 10 L3 cache misses, while no L3 cache misses result from the pair of operations in a local share.

We measured the L3 misses for the Linux file descriptor microbenchmark described in Section 1. On one core a duplicate and close incurs no L3 cache misses; however, L3 misses increase with the number of cores and at 16 cores a `dup` and `close` costs 51 L3 cache misses.

A Corey application can use shares to avoid bottlenecks that might be caused by contention on kernel data structures when creating and manipulating kernel objects. A Linux application, on the other hand, must use sharing policies specified by kernel implementers, which



(a) Corey and Linux performance.



(b) Corey improvement over Linux.

Figure 10: MapReduce `wri` results.

might force inter-core sharing and unnecessarily limit scalability.

8.5 Applications

To demonstrate that address ranges, kernel cores, and shares have an impact on application performance we presents benchmark results from the Corey port of Metis and from `webd`.

8.5.1 MapReduce

We compare Metis on Corey and on Linux, using the `wri` MapReduce application to build a reverse index of the words in a 1 Gbyte file. Metis allocates 2 Gbytes to hold intermediate values and like `memclone`, Metis allocates cores from chips in a round-robin fashion.

On Linux, cores share a single address space. On Corey each core maps the memory segments holding intermediate results using per-core shared address ranges. During the map phase each core writes $\langle \text{word}, \text{index} \rangle$ key-value pairs to its per-core intermediate results memory. For pairs with the same word, the indices are grouped as an array. More than 80% of the map phase is spent in `strcmp`, leaving little scope for improvement from address ranges. During the reduce phase each core copies all the indices for each word it is responsible for from the intermediate result memories of all cores to a freshly allocated buffer. The reduce phase spends most of its time copying memory and handling soft page faults. As the number of cores increases, Linux cores contend while manipulating address space data while Corey's address ranges keep contention costs low.

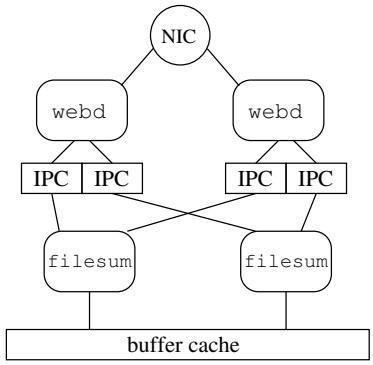


Figure 11: A `webd` configuration with two front-end cores and two `filesum` cores. Rectangles represent segments, rounded rectangles represents pcores, and the circle represents a network device.

Figure 10(a) presents the absolute performance of `wri` on Corey and Linux and Figure 10(b) shows Corey's improvement relative to Linux. For less than eight cores Linux is 1% faster than Corey because Linux's soft page fault handler is about 10% faster than Corey's when there is no contention. With eight cores on Linux, address space contention costs during reduce cause execution time for reduce to increase from 0.47 to 0.58 seconds, while Corey's reduce time decreases from 0.46 to 0.42 seconds. The time to complete the reduce phase continues to increase on Linux and decrease on Corey as more cores are used. At 16 cores the reduce phase takes 1.9 seconds on Linux and only 0.35 seconds on Corey.

8.5.2 `Webd`

As described in Section 6.2, applications might be able to increase performance by dedicating application data to cores. We explore this possibility using a `webd` application called `filesum`. `Filesum` expects a file name as argument, and returns the sum of the bytes in that file. `Filesum` can be configured in two modes. "Random" mode sums the file on a random core and "Locality" mode assigns each file to a particular core and sums a requested file on its assigned core. Figure 11 presents a four core configuration.

We measured the throughput of `webd` with `filesum` in Random mode and in Locality mode. We configured `webd` front-end servers on eight cores and `filesum` on eight cores. This experiment did not use a kernel core or device polling. A single core handles network interrupts, and all cores running `webd` directly manipulate the network device. Clients only request files from a set of eight, so each `filesum` core is assigned one file, which

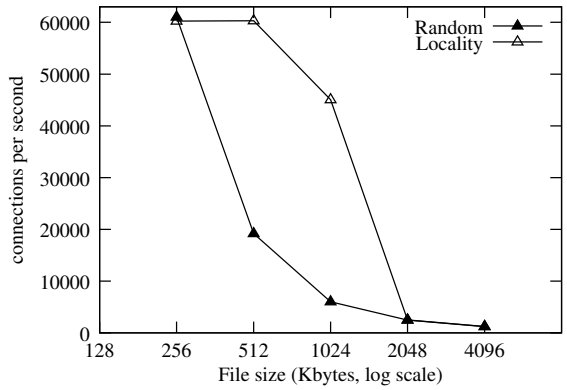


Figure 12: Throughput of Random mode and Locality mode.

it maps into memory. The HTTP front-end cores and `filesum` cores exchange file requests and responses using a shared memory segment.

Figure 12 presents the results of Random mode and Locality mode. For file sizes of 256 Kbytes and smaller the performance of Locality and Random mode are both limited by the performance of the `webd` front-ends' network stacks. In Locality mode, each `filesum` core can store its assigned 512 Kbyte or 1024 Kbyte file in chip caches. For 512 Kbyte files Locality mode achieves 3.1 times greater throughput than Random and for 1024 Kbyte files locality mode achieves 7.5 times greater throughput. Larger files cannot be fit in chip caches and the performance of both Locality mode and Random mode is limited by DRAM performance.

9 DISCUSSION AND FUTURE WORK

The results above should be viewed as a case for the principle that applications should control sharing rather than a conclusive "proof". Corey lacks many features of commodity operating systems, such as Linux, which influences experimental results both positively and negatively.

Many of the ideas in Corey could be applied to existing operating systems such as Linux. The Linux kernel could use a variant of address ranges internally, and perhaps allow application control via `mmap` flags. Applications could use kernel-core-like mechanisms to reduce system call invocation costs, to avoid concurrent access to kernel data, or to manage an application's sharing of its own data using techniques like computation migration [6, 14]. Finally, it may be possible for Linux to provide share-like control over the sharing of file descriptors, entries in buffer and directory lookup caches, and network stacks.

10 RELATED WORK

Corey is influenced by Disco [5] (and its relatives [13, 30]), which runs as a virtual machine monitor on a NUMA machine. Like Disco, Corey aims to avoid kernel bottlenecks with a small kernel that minimizes shared data. However, Corey has a kernel interface like an exokernel [9] rather than a virtual machine monitor.

Corey's focus on supporting library operating systems resembles Disco's SPLASHOS, a runtime tailored for running the Splash benchmark [26]. Corey's library operating systems require more operating system features, which has led to the Corey kernel providing a number of novel ideas to allow libraries to control sharing (address ranges, kernel cores, and shares).

K42 [3] and Tornado [10], like Corey, are designed so that independent applications do not contend over resources managed by the kernel. The K42 and Tornado kernels provide clustered objects that allow different implementations based on sharing patterns. The Corey kernel takes a different approach; applications customize sharing policies as needed instead of selecting from a set of policies provided by kernel implementers. In addition, Corey provides new abstractions not present in K42 and Tornado.

Much work has been done on making widely-used operating systems run well on multiprocessors [4]. The Linux community has made many changes to improve scalability (e.g., Read-Copy-Update (RCU) [20] locks, local runqueues [1], `libnuma` [16], improved load-balancing support [17]). These techniques are complementary to the new techniques that Corey introduces and have been a source of inspiration. For example, Corey uses tricks inspired by RCU to implement efficient functions for retrieving and removing objects from a share.

As mentioned in the Introduction, Gough *et al.* [12] and Veal and Foong [29] have identified Linux scaling challenges in the kernel implementations of scheduling and directory lookup, respectively.

Saha *et al.* have proposed the Multi-Core Run Time (McRT) for desktop workloads [23] and have explored dedicating some cores to McRT and the application and others to the operating system. Corey also uses spatial multiplexing, but provides a new kernel that runs on all cores and that allows applications to dedicate kernel functions to cores through kernel cores.

An Intel white paper reports use of spatial multiplexing of packet processing in a network analyzer [15], with performance considerations similar to `webd` running in Locality or Random modes.

Effective use of multicore processors has many aspects and potential solution approaches; Corey explores only a few. For example, applications could be assisted in coping with heterogeneous hardware [25]; threads could be automatically assigned to cores in a way that minimizes cache misses [27]; or applications could schedule work in order to minimize the number of times a given datum needs to be loaded from DRAM [7].

11 CONCLUSIONS

This paper argues that, in order for applications to scale on multicore architectures, applications must control sharing. Corey is a new kernel that follows this principle. Its address range, kernel core, and share abstractions ensure that each kernel data structure is used by only one core by default, while giving applications the ability to specify when sharing of kernel data is necessary. Experiments with a MapReduce application and a synthetic Web application demonstrate that Corey's design allows these applications to avoid scalability bottlenecks in the operating system and outperform Linux on 16-core machines. We hope that the Corey ideas will help applications to scale to the larger number of cores on future processors. All Corey source code is publicly available.

ACKNOWLEDGMENTS

We thank Russ Cox, Austin Clements, Evan Jones, Emil Sit, Xi Wang, and our shepherd, Wolfgang Schröder-Preikschat, for their feedback. The NSF partially funded this work through award number 0834415.

REFERENCES

[1] J. Aas. Understanding the Linux 2.6.8.1 CPU scheduler, February 2005. `http://josh.trancesoftware.com/linux/`.

[2] A. Agarwal and M. Levy. Thousand-core chips: the kill rule for multicore. In *Proceedings of the 44th Annual Conference on Design Automation*, pages 750–753, 2007.

[3] J. Appavoo, D. D. Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, J. Xenidis, M. Stumm, and L. Soares. Experience distributing objects in an SMMP OS. *ACM Trans. Comput. Syst.*, 25(3):6, 2007.

[4] R. Bryant, J. Hawkes, J. Steiner, J. Barnes, and J. Higdon. Scaling linux to the extreme. In *Proceedings of the Linux Symposium 2004*, pages 133–148, Ottawa, Ontario, June 2004.

[5] E. Bugnion, S. Devine, and M. Rosenblum. DISCO: running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 143–156, Saint-Malo, France, October 1997. ACM.

[6] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[7] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling Threads for Constructive Cache Sharing on CMPs. In *Proceedings of the 19th ACM Symposium on Parallel Algorithms and Architectures*, pages 105–115. ACM, 2007.

[8] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[9] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, December 1995. ACM.

[10] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 87–100, 1999.

[11] Google Performance Tools. `http://goog-perftools.sourceforge.net/`.

[12] C. Gough, S. Siddha, and K. Chen. Kernel scalability—expanding the horizon beyond fine grain locks. In *Proceedings of the Linux Symposium 2007*, pages 153–165, Ottawa, Ontario, June 2007.

[13] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 154–169, Kiawah Island, SC, October 1999. ACM.

[14] W. C. Hsieh, M. F. Kaashoek, and W. E. Weihl. Dynamic computation migration in dsm systems. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, Washington, DC, USA, 1996.

[15] Intel. Supra-linear Packet Processing Performance with Intel Multi-core Processors. `ftp://download.intel.com/technology/advanced_comm/31156601.pdf`.

[16] A. Klein. An NUMA API for Linux, August 2004. `http://www.firstfloor.org/~andi/numa.html`.

[17] Linux kernel mailing list. http://kerneltrap.org/node/8059.

[18] Linux Symposium. http://www.linuxsymposium.org/.

[19] lwIP. http://savannah.nongnu.org/projects/lwip/.

[20] P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell. Read-copy update. In *Proceedings of the Linux Symposium 2002*, pages 338–367, Ottawa, Ontario, June 2002.

[21] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[22] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, pages 13–24. IEEE Computer Society, 2007.

[23] B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang. Enabling scalability and performance in a large-scale CMP environment. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 73–86, New York, NY, USA, 2007. ACM.

[24] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable Locality-Conscious Multithreaded Memory Allocation. In *Proceedings of the 2006 ACM SIGPLAN International Symposium on Memory Management*, pages 84–94, 2006.

[25] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems (MMCS)*, Boston, MA, USA, June 2008.

[26] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44.

[27] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 47–58, New York, NY, USA, 2007. ACM.

[28] uClibc. http://www.uclibc.org/.

[29] B. Veal and A. Foong. Performance scalability of a multi-core web server. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 57–66, New York, NY, USA, 2007. ACM.

[30] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Proceedings of the 7th international conference on Architectural support for programming languages and operating systems*, pages 279–289, New York, NY, USA, 1996. ACM.

[31] K. Yotov, K. Pingali, and P. Stodghill. Automatic measurement of memory hierarchy parameters. *SIGMETRICS Perform. Eval. Rev.*, 33(1):181–192, 2005.

CuriOS: Improving Reliability through Operating System Structure

Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, Roy H. Campbell
University of Illinois at Urbana-Champaign
{*fdavid,emchan,jcarlyle,rhc*}*@illinois.edu*

Abstract

An error that occurs in a microkernel operating system service can potentially result in state corruption and service failure. A simple restart of the failed service is not always the best solution for reliability. Blindly restarting a service which maintains client-related state such as session information results in the loss of this state and affects all clients that were using the service. CuriOS represents a novel OS design that uses lightweight distribution, isolation and persistence of OS service state to mitigate the problem of state loss during a restart. The design also significantly reduces error propagation within client-related state maintained by an OS service. This is achieved by encapsulating services in separate protection domains and granting access to client-related state only when required for request processing. Fault injection experiments show that it is possible to recover from between 87% and 100% of manifested errors in OS services such as the file system, network, timer and scheduler while maintaining low performance overheads.

1 Introduction

Operating system reliability has been studied for several decades [39, 19, 34, 46], but remains a major concern today [47]. Operating system errors can be caused by both hardware and software faults. Hardware faults can arise due to various factors, some of which are aging, temperature, and radiation-induced bit-flips in memory and registers (Single Event Upsets [30]). Software faults (bugs) are also very common in large and complex operating systems [13].

In the past, designs for reliable computer systems have used redundancy in hardware and OS software to attempt recovery from errors [5, 6]. Redundancy can mask transient and permanent hardware faults as well as some software faults [4]. However, it does not address the insidious problem of the propagation of undetected errors [34]. Additionally, these systems are extremely expensive to build and use [44].

Errors in a monolithic OS can easily propagate and corrupt other parts of the system [22, 52], making recovery extremely difficult. Microkernel designs componentize the OS into servers managed by a minimal kernel. These servers provide functionality such as the file system, networking and timers. User applications and other OS components are modeled as clients of these servers. Inter-component error propagation is significantly reduced because, in many microkernel designs, servers usually execute in their own restricted address spaces similar to user processes [25, 37].

Recovery from a microkernel server failure is typically attempted by restarting it. The intuition behind this approach is that reinitializing data structures from scratch by restarting a server usually fixes a transient fault. This is similar to microrebooting [10]. In Minix3 [25], for example, server restarts are performed by the *Reincarnation Server* [47]. If the server managing a printer crashes, it causes a temporary unavailability of the printer until it is restarted. Unfortunately, this approach to recovery does not always work. Many OS services maintain state related to clients. In such cases, a server restart results in the loss of this state information and affects all clients that depend on the server. For example, a failure of the file system server in Minix3 impacts all clients that were using the file system. Simply restarting the file system server does not prevent errors from occurring in these existing clients. Reads and writes to existing open files cannot be completed because the restarted server cannot recognize the file handles that are presented to it. Thus, while stateless servers such as some device drivers can be restarted to recover the system, this technique is not applicable for many important OS services that manage client-related state.

Writing clients to take into account OS service restarts and state loss is a possible solution. This requires clients to subscribe to server failure notifications and can re-

sult in increased code complexity. Another possible solution is to provide some form of persistence to the server's client-related state information. This allows a restarted server to continue processing requests from existing clients. Some microkernel operating systems like Chorus and Minix3 support the ability to persist state in memory through restarts; but they do not use this functionality for OS servers and, currently, only provide it as a service for user applications or device drivers.

Attempts to solve the state loss problem by simply persisting server state across a restart do not address the possible corruption of this state due to error propagation. An error that occurs in an OS server, like a typical software error, can potentially corrupt any part of its state [27] before being detected. This highlights yet another significant limitation of traditional microkernel systems. While such systems minimize inter-component error propagation, nothing prevents intra-component error propagation.

Checkpointing OS service state in order to mitigate the effects of error propagation is not a viable solution because rolling back to a consistent system state requires checkpointing of client state as well. Additionally, multiple checkpoints may have to be maintained in order to avoid rolling back to an incorrect state. This may be expensive in terms of memory and performance.

In this paper, we present CuriOS, which adopts an approach that significantly minimizes error propagation between as well as within OS services and recovers failed services transparently to clients. We accomplish this by lightweight distribution, isolation and persistence of client-specific state information used by OS servers. Client-specific state is stored in client-associated, but client-inaccessible memory and servers are only granted access to this information when servicing a request. Because this state is not associated with the server, it persists after a server restart. This distribution of state

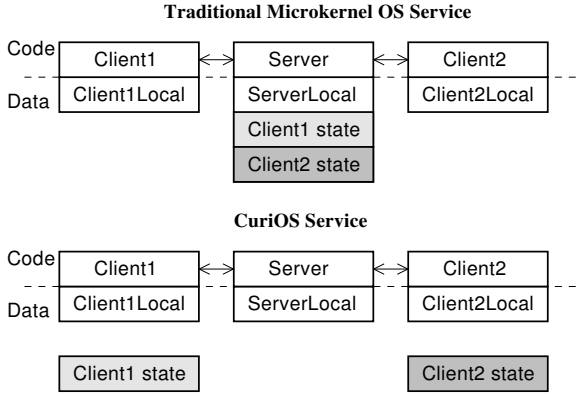


Figure 1: State Distribution

is illustrated in figure 1. A server failure that occurs when servicing a client can only affect that client and the restarted server can continue to process other requests normally.

Distribution of state information from servers to clients for fault tolerance is not new. Researchers have exploited this technique to improve the reliability of file system services in distributed operating systems such as Sprite [53] and Chorus/MiX [33]. A more widely known example is Sun's stateless Network File System (NFS) [41]. But these designs do not protect the state information from being manipulated by clients and leads to various security problems such as those with NFS [51, 32]. Our design supports safe distribution of state by protecting the state from modification by clients. Our implementation is also lightweight because we use virtual memory remapping instead of memory copying to grant access to state. Additionally, we provide a generic framework for implementing distributed state and recovery for any OS service, not just the file system.

CuriOS is written in C++ and is based on the Choices object-oriented operating system [8]. It is being developed to provide a highly reliable OS environment for mobile devices such as cellular phones powered by an ARM processor.

Our work is complementary to other research in OS error detection such as the language-based type-safety techniques used in SafeDrive [55] and software guards used in the XFI system [50]. Employing such techniques in CuriOS can improve error detection latency and further reduce error propagation.

A preliminary design for CuriOS is available in a previous publication [18]. The contributions of this paper include:

1. A comparison and analysis of the effect of memory errors on OS services of several popular microkernel architectures, some of which are designed for reliability.

2. A detailed description of the state management framework implementation in CuriOS that reduces intra-component error propagation and enables transparent OS service recovery.

3. An evaluation of the CuriOS design using fault-injection experiments performed on several OS services.

The remainder of this paper is organized as follows. We investigate several related operating systems in Section 2. In Section 3, we look at the results of our investigations and present our observations for an operating system design that supports transparent recovery. Section 4 presents a brief introduction to the CuriOS architecture and details the framework used to manage OS

service state information. Section 5 describes the design and implementation of a few transparently restartable CuriOS components and drivers. We present an evaluation of several aspects of our current implementation in 6. We discuss a number of additional related topics in Section 7 and conclude in Section 8. In this paper, we limit our scope to the reliability aspects of CuriOS. A short discussion of some security issues is available in Section 7.

The dependability related terms used in this paper conform to the taxonomy suggested by Avizienis et al [3]. All names that use the font `Class` represent C++ classes.

2 Related Operating Systems

Some microkernel operating systems that are closely related to our work are Minix3 [25], L4 [37], Chorus [40] and EROS [43]. For the evaluation of each of these microkernel-based operating systems, we manually inject memory access errors into different OS servers to explore the effect of an OS error on its reliability. A memory access error is the typical manifestation of a hardware or software fault in an OS [52].

In all our experiments, a memory access error results in the termination of the OS server. Table 1 shows the results of our experiments. The effect of server termination after encountering the memory access error is shown in the third column. The last column presents our analysis of whether a restarted server will continue serving existing clients correctly (if restartability support were included in the corresponding OS). Except for Minix3, which already implements restartable services, this observation is based purely on source code analysis. Brief explanations for our conclusions are provided in each row. The entries in the last column for Minix3 are actual experimental results.

The rest of this Section discusses reliability aspects of the previously mentioned systems and several other related operating systems in more detail.

2.1 Minix3

Reliability support in Minix3 is provided by the *Reincarnation Server* which is able to restart both failed services and device drivers. Server restarts work well only for device drivers [47, 24]. This is substantiated by our experiments (Table 1). The file system server crashes on all invalid memory accesses and results in an unusable system. Even if the file system server were restarted correctly, existing open files would be inaccessible because of the lost server state.

Minix3 includes a data store server that can be used to store state that persists after a failure induced restart. The Minix3 data store provides some protection from errors in a server because it resides in a separate address space from the server. It has been used to implement failure resilience for device drivers [26]. A drawback of the data store approach is the additional communication and data copying overhead involved. This approach also does not restrict intra-component error propagation.

2.2 L4/Iguana

Iguana [35] is a suite of OS services that are implemented for the L4 microkernel [37]. This comprises basic OS services such as naming, memory management, timer and some device drivers. Our experiments study the behavior of some Iguana services when they encounter memory errors. Unlike Minix3, there isn't any support for restartable services. An analysis of the source code shows that server restartability, if implemented, still does not solve the problem of preventing the corruption of state and recovering it. As an example, the Iguana timer service maintains information about clients to which it periodically sends messages. This information will be irrecoverably lost upon a restart. A stateless server like the serial driver, on the other hand, can be restarted and may continue to work for existing clients.

More complex functionality such as a file system is part of the L4Linux [23] suite, which implements a complete Linux system as a user-mode server. Since most of the functionality required by Linux applications is implemented in this server, the reliability of all L4Linux applications depends on the reliability of this server, and thus, this design is not any more reliable than the normal monolithic Linux OS. This has been improved to some extent by isolating device drivers in separate virtual L4Linux servers [36].

2.3 Chorus

The Chorus OS [40] is designed for high reliability and is used in several telecommunication systems. In contrast to Minix3 and L4, services are executed in privileged mode and share the same address space as the microkernel. Chorus includes "Hot Restart" technology [1] that allows servers to maintain state in persistent memory and resume execution quickly after a failure. Unlike both the design we use in CuriOS and the Minix3 data store, all allocated persistent memory in Chorus is permanently mapped into the server domain. There is no mechanism in place that prevents state information saved in the allocated persistent memory from being potentially corrupted by an error that occurs in a server. Unfortunately, Chorus' operating system services do not take advantage of the "Hot Restart" functionality.

Table 1: Microkernel Operating System Recoverability after Server Failures

μkernel	Failed Server	Immediate Effect	After Restart
Minix3	File System (fs)	System unusable.	× Server is not restarted because the *Reincarnation Server* depends on the file system. Also, all current file system state information is lost.
	Network (inet)	All existing network connections fail.	× Restart does not help re-establish connections because state information is lost.
	Random Numbers (random)	Temporary read failure.	✓ Once the server is restarted, client reads begin working again.
	Printer Driver (printer)	Temporary printer access failure.	✓ Print job completes successfully after spooler retries request to the restarted printer server.
L4	Timer (ig_timer)	System unusable.	× All clients stop receiving timer interrupts. Restart does not help because clients waiting on interrupts can't re-register.
	Name Server (ig_naming)	No immediate effect.	× But many critical services inaccessible because lookup of registered names fail. Restart does not help because all registered clients need to re-register.
	Serial (ig_serial)	Serial port inaccessible.	✓ Request retries will eventually work.
Chorus	File System (vfs)	System unusable.	× Restart does not help because file system state information is lost.
	Network (netinet)	System unusable.	× Restart does not help recover existing network connections.
	Timer (kern)	System unusable.	× Restart does not address clients waiting on timeout.
EROS	Memory allocator (spacebank)	System unusable.	✓ Restore from a previous checkpoint may fix this error.
	Process Creator	System cannot create new processes.	✓ Restore from a previous checkpoint may fix this error.

2.4 EROS

EROS [43] is a capability-based system which saves periodic snapshots of the entire machine state to disk. When the system recovers after a crash, the last written snapshot is reloaded. This approach only works when the error is not present in the snapshot. Though the system performs some consistency checks on snapshots, correctness cannot be assured and several previous snapshots may have to be reloaded before a working version is obtained. Minix3's approach of restarting an erroneous server results in a re-creation of all internal state and has better chances of eliminating errors. Another drawback is that snapshots of large systems and device state (not currently performed by EROS) can be expensive in terms of memory and performance. Reverting to a previous system snapshot on a failure also results in a loss of all work done since the snapshot. This may be undesirable in some situations. For example all user input since the last snapshot is lost.

2.5 Other Systems

The Exokernel OS architecture [21] places most operating system abstractions in an application library and securely multiplexes machine resources. Similar to a monolithic kernel, error propagation is possible throughout the library OS and the application. There is no mechanism that provides transparent recovery for an application when errors occur in the associated library OS. An important advantage of the exokernel approach is that errors only affect the process in which they occur. This benefit is at the cost of a complex design for multiplexing shared resources like the storage subsystem. Four design iterations were required to build the XN storage system [31]. The Nemesis OS [29] also adopted a vertical structure similar to the exokernel architecture while providing explicit low-level guarantees for reserved resources. Error propagation was limited by enforcing isolation between device driver, system and application domains. The design of Nemesis was driven by QoS considerations and not surprisingly, does not include recov-

ery support for arbitrary errors in components. However, Nemesis provides QoS isolation between the clients of a system service. Services are designed to prevent one client from adversely affecting the QoS observed by others.

The Singularity system [28] adopts a radically different approach to security and reliability by using software enforcement of address spaces. CuriOS relies on hardware support to enforce memory protection.

3 Observations

From our study of the operating systems in the previous Section, we are able to make several observations about how the design of an operating system can impact its ability to transparently recover in the event of the failure and restart of an OS service.

Transparency of addressing: Clients should be able to use the same address to access the OS service after it is restarted. In EROS, since the whole system is restored to a previous checkpoint, this property is true. This is not supported by Chorus, whose hot restart algorithm restarts servers with a new address. Nor is this supported by L4 or Minix3 since a restarted server would be assigned a different address. A name server can be used to ameliorate this problem by maintaining a consistent name for the server across a restart. The restarted server would register its new address with the name server to provide continued availability.

Minix3 achieves transparency of addressing to some degree by using the file system server as a name server. A server can register itself as the handler for a device entry on the file system. For instance, the Minix3 *random* server mentioned in table 1 handles requests for the */dev/random* file system entry. In our experiment, we opened */dev/random* using the *open* system call and used the returned file handle to read a stream of random numbers from the server. If the *random* server crashed, reads using this file handle failed; however, once the server was restarted, reads using the same handle began to work once again.

Suspension of clients for duration of recovery: Clients should not time out or initiate new requests during the recovery phase. This property is supported by Chorus. In Minix3, clients are allowed to run when the server is restarting, and this results in errors when a client attempts to communicate with it. This is also the case in L4; the client will receive an error when it tries to communicate with a server that may be restarting. The whole system is restored to a previous checkpoint in EROS and therefore, this property is not applicable.

Persistence of client-related state: When a service is restarted, requests from clients must not fail because the server lost client-related state. Client-related state must be preserved and made available to the restarted server. Chorus and Minix3 have some support for in-memory state preservation, but this is not exploited by any of the OS services they support. An alternative is to save this information to stable storage. In EROS, all computation since the last saved checkpoint is lost.

Isolation of client-related state: Designs of existing microkernel operating systems provide unrestricted access to client-related state within a server. An error that occurs in the server can potentially corrupt state related to all clients. This intra-component error propagation problem exists in a large number of important microkernel OS services. In EROS, error propagation may lead to inconsistent data being checkpointed.

In the next Section, we describe how CuriOS fulfills all of these requirements and enables transparently-restartable OS components.

4 CuriOS Design

4.1 Structure and Overview

CuriOS is structured as a collection of interacting objects that represent various components and services. An object can be confined to an isolated memory protection domain in order to reduce error propagation. We refer to such an object as a *protected object* (PO). All methods on a protected object are executed with reduced privileges and run with hardware enforced memory protection. CuriOS applies the principle of least privilege to protected objects and only grants them access to memory regions that are required for correct operation. This prevents an error that occurs while running code in a protected object from corrupting other parts of the system by overwriting memory outside of the protected object. The reduced privilege execution mode also prevents protected object code from executing privileged processor instructions. Devices can be made accessible from within protected objects in order to encapsulate device drivers.

A protected object in CuriOS is analogous to a "server" in a traditional microkernel system. Our implementation of protected objects on the ARM platform only enforces restrictions on memory access. Implementations of protected objects on other platforms such as the x86 can additionally exploit architectural features to provide access control for other resources such as IO ports.

Protected objects work together with a small kernel, known as *CuiK*, in order to provide standard OS services as shown in figure 2. CuiK is a thin layer of the OS that

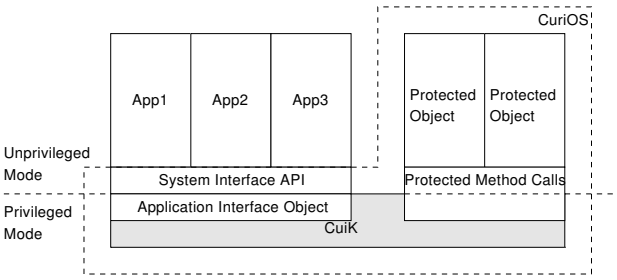


Figure 2: CuriOS Organization

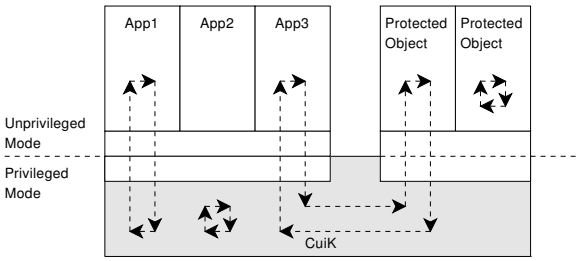


Figure 3: CuriOS Threads

runs with the highest privileges. It is composed of a small set of objects that manage low level architecture specific functionality such as interrupt dispatching and context switching. Communication between protected objects is managed by CuiK.

CuiK uses *protected method calls* to invoke operations on protected objects. Each protected object is assigned a private heap. A private stack is reserved for every thread that accesses the protected object. This stack is allocated at the first invocation of a protected method, contributing to a small delay in processing the first call to a protected object. Subsequent invocations of protected methods on the same protected object by the same thread reuse this stack. A protected method call results in a switch to a reduced privilege execution mode and constrained access rights to memory. The private stack and the heap are mapped in with read-write privileges. The rest of CuriOS is mapped in with read-only privileges. Permissions to write to any additional memory has to be explicitly granted by CuiK.

Our current implementation of protected method calls uses a wrapper object that intercepts method calls to a protected object and manages memory access control, processor mode switching and recovery. The combination of protected objects and CuiK results in a single address space operating system [11], where virtual addresses are identical across various components, but access permissions differ.

Threads in CuriOS are managed by CuiK. Using defined interfaces, a thread executing in CuriOS can cross user-space application, kernel, and protected object boundaries. For example, a system call in an application causes the thread to cross from user-space into the CuiK kernel. This same thread can cross from CuiK into a protected object using a protected method call. Some example threads are illustrated in figure 3.

CuriOS is written in C++ and uses object-oriented techniques to minimize code duplication and improve portability. Wrapper classes, for example, inherit from a common base class that provides the support functions used to switch protection domains and manage private heaps and stacks. In our current implementation, wrapper classes use multiple inheritance and also inherit from the class representing the object being wrapped. This allows the wrapper to exploit polymorphism and substitute the protected object anywhere in the system.

C++ exception handling is used as the error signaling mechanism in CuriOS [17]. Exceptions are raised for both processor signaled errors such as invalid memory accesses and for externally signaled errors such as OS infinite loop lockups (signaled by a watchdog timer) [16].

Exceptions are raised when errors are detected while executing code within a protected object. Exceptions that are not handled within the object are intercepted at the wrapper which attempts to destroy and re-create the protected object. The wrapper maintains a copy of the constructor arguments (if any) in order to re-create the protected object. This is similar to microrebooting or server restarts in Minix3 and can be used to fix transient hardware or software faults. The protected object is re-created in-place in memory ensuring that external references to it remain valid. This provides *transparency of addressing*. The method call is immediately retried on the newly constructed protected object. Multiple retry failures cause an exception to be returned to the caller. All normal system activity is suspended until the recovery is completed. Thus *clients are suspended for the duration of recovery*.

4.2 Server State Management

A server providing an OS service is implemented using a protected object. Clients are either user applications, or other protected objects. A protected object that represents an OS service can in turn operate as a client to another server.

A server that needs to maintain state information about clients uses state management functionality provided by CuiK to distribute, isolate, and persist client-related state. Servers that are completely stateless can be easily restarted and do not require this functionality.

A *Server State Region* (SSR) is an object represent-

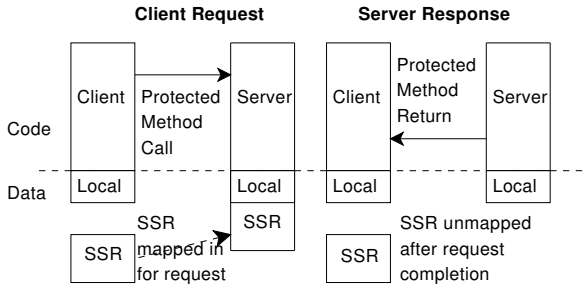


Figure 4: Request Processing

ing a region of memory that is allocated to store an OS server's client-related information. An SSR is created when a client establishes a connection to the server. For accounting purposes, the memory associated with the SSR is charged to the client. SSRs are protected from both the server and the client through hardware-supported virtual memory protection mechanisms. A client is never granted access to its SSR. A server is only granted write access to a client's SSR when it is processing a request from that client (see figure 4). The SSR is passed as an argument to the server's protected method call. The server can then use the SSR to store client-related information. It has full control over management of the memory within the SSR. Write permissions to the SSR are revoked when the protected method call returns. SSRs are implemented using a C++ object that holds a pointer to a hardware-protectable region of memory.

All SSRs in CuriOS are managed by a singleton object called the `SSRManager`. The `SSRManager` provides the functions used to register a new server, bind a client to a server (resulting in the creation of an SSR), undo a client-server binding (deletion of the associated SSR) and enumerate all the SSRs associated with a server. Each server using SSRs is required to provide a recovery routine that is invoked immediately after the server object is re-created upon a failure. This routine can query the `SSRManager` to obtain all associated SSRs in order to re-create the internal state of the restarted server.

The SSR-based state management framework provides *persistence and isolation of client-related state*.

4.3 OS Service Construction

How should the state of a generic OS server be structured in order to use the state management support provided by CuiK? There are two types of stateful servers. The first type is a server that does not require collective information about all of its clients in order to service a request. A server that provides pseudo random numbers based on a per-client seed is one such example. It only needs to know one client's seed in order to service a request from that client. Such servers can store client information in SSRs and can be transparently restarted upon a failure. All future client requests will continue to work correctly because its SSRs and the information stored in them is not lost.

The second type is a server that requires knowledge about all of its clients in order to service a request. Examples of this type are OS services like the scheduler and timer managers. Such servers can store client related information in SSRs and can redundantly cache this information locally to process requests. Upon a restart, such a server should be able to re-create its internal state from its distributed SSRs.

Many CuriOS components and drivers are stateless or structured as one of these two types of servers. When restarting, the server's recovery routine re-creates internal state from all SSRs. It is possible that the SSR that was in use at the time an error occurs is corrupted. The recovery routine can check the consistency of the objects in SSRs using simple heuristics before using them. CuriOS uses magic numbers in objects and these can be checked for corruption. We also use server-specific checks to ensure that pointers and numbers are within expected ranges. Unlike EROS which does consistency checks of all state during normal running time, these SSR consistency checks in CuriOS are only performed on exceptional conditions that require server recovery.

CuriOS servers can be multi-threaded and our current implementation shares the same virtual memory mappings for all threads. Thus, it is possible that multiple SSRs are mapped in when an error occurs. In this case, the error propagation is limited to the SSRs that are currently mapped in. This can be further improved by including support for thread-level protection.

4.4 Recoverable Errors

Protected method calls to servers are designed to retry the request after a server fails and restarts during the processing of the request. SSR-based recovery addresses a large class of errors that result in corrupted local OS service state. Complete reconstruction of service local state during recovery can remedy any such corruption. When an SSR is corrupted and multiple attempts at recovery fail, only the client associated with the corrupted SSR is affected and will need to be notified of a failure. The other clients of the service can continue to function normally. Thus, SSR-based recovery can minimize the impact of a software bug that is triggered by a specific client request and a consequent failure. When repeated attempts at processing the request fail, the client can be notified and the service can continue processing other requests that do not trigger the bug.

Restarting a service that has visible external effects may not always result in correct behavior. For example, restarting a printer driver due to a failure may cause another copy of the print job to be dispatched. This problem may be ameliorated to some degree by writing code that is restart-aware. This is achieved by incorporating some means of recording the progress made in servicing a request. This limitation has also been acknowledged for device driver restarts in Minix3 [26]. Similar to the approach taken by Minix3, we advocate notification of possible non-transparent recovery to applications or users.

5 CuriOS Services

Timer Management: A `PeriodicTimerManager` service provided by CuriOS allows user applications to access timer functionality. There is only one instance of this class in the system and it is created as a PO. Clients can start a timer by placing a request to be notified periodically. The job of the `PeriodicTimerManager` is to periodically signal a semaphore that the client waits upon. In order to support recovery from a restart of the `PeriodicTimerManager` service, SSRs are used to persist and distribute information regarding each client. The timer period, starting time and semaphore are stored in every client's SSR. The `PeriodicTimerManager` is implemented using a linked list of pending client notifications. Upon a failure-induced restart, the `PeriodicTimerManager` can re-create this complete internal linked list from the timer period and starting time information in the distributed SSRs.

Scheduling: CuriOS schedulers are modeled as process containers which manage a collection of processes and provide a scheduling strategy by presenting a method to pick the next process to run. A FIFO scheduler, for example, is implemented using a linked list of processes. The scheduler is created as a PO with clients as individual processes. A client SSR includes the pointer to the corresponding `Process` object and scheduling strategy-specific information such as priorities. If the scheduler is restarted after a failure, it queries the `SSRManager` for all its clients and re-creates its internal list.

Networking: The recovery mechanisms in CuriOS allow for the construction of an extremely reliable network stack. CuriOS uses the LWIP networking stack [20] encapsulated in two restartable protected objects: one for managing TCP connections and the other for UDP. LWIP creates a *tcp_pcb* or a *udp_pcb* data structure to manage state information for every connection. We refactored LWIP code to place these data structures within SSRs. In the case of TCP, for example, this includes all information necessary to service the incoming and outgoing packets of a connection. This includes the network addresses, ports, windows, sequence numbers and so on. If the TCP service crashes and is restarted, this information is used to resume the processing of packets. If this state information is not preserved during a restart, the unfortunate consequence is that all network connections in progress will be terminated.

Each SSR is associated with a client `Socket` object and is mapped into the TCP PO's address space when interacting with it. When there is an incoming packet, the corresponding SSR is located and mapped in before sending it through the stack. A similar approach is used to provide access to SSRs for the TCP PO's timer driven events.

All the assert code in LWIP was converted to throw exceptions instead of halting the stack. This comprehensive error detection in LWIP helps reduce error propagation and improves recovery rates.

File Systems: CuriOS currently supports two different file systems. `CramFSFileObject` is a class that provides access to a compressed file on the read-only CramFS file system [14]. When a file is opened, an instance of this class is created as a PO. This instance only has information about its backing storage and does not maintain any state regarding clients. Hence it does not require usage of the server state management functionality. The method call to read a file provides both the offset into the file and the required number of bytes. The PO is only granted privileges to modify its own data and the destination buffer. Calls to other objects like the backing storage are mediated by CuiK. Using a PO for each file has several reliability benefits. An error that occurs when processing one file is contained within the PO and cannot corrupt arbitrary memory in the system. If the error were transient, a restarted PO can continue serving clients. If there is an error in a compressed file stored on the disk that causes the decompression routines to fail, it only causes an error in the clients that were reading that particular file.

CuriOS also includes support for the Linux ext2 file system. An `Ext2Container` PO is created for every ext2 file system on disk. This manages the inode and free space bitmaps. If this PO crashes and restarts, it can re-read this information from disk. An `Ext2Inode` PO is tasked with managing all interaction with a file. This PO only has privileges to modify the inode it represents, which, in turn, has all the pointers to disk blocks comprising the file. This has similar reliability benefits as the `CramFSFileObject` protected object. Since POs are re-created in-place, the same objects can be used to access the file after the service is restarted.

It is important to realize that when we refer to file system service recovery, we are not dealing with recovering corrupted file system state on disk. There are many other programs that are designed to handle corrupted data on disk and this is an orthogonal problem with recovering the state information maintained in a live file system server.

Device Drivers: The serial port driver in CuriOS is implemented as a PO with complete access to the memory mapped registers of the serial port controller. This PO is stateless and only has one client: the CuriOS console object. Errors that occur when reading or writing to the serial port are handled by restarting this PO and retrying the request. CuriOS has a NOR flash driver that is implemented as a stateless PO. This PO is created with read/write access rights to physical memory regions that map to NOR flash chips. An error that occurs in this PO can lead to potential corruption of arbitrary data stored in NOR flash, but cannot easily corrupt read-only mapped system memory. Protected objects are also used to encapsulate drivers for interacting with the hardware timers. These are used to start, query and stop the hardware timers. Interrupts from the hardware timers are also dispatched to them. These are currently stateless and can be restarted. The `PeriodicTimerManager` service depends on the correct functioning of these driver objects.

6 Evaluation

CuriOS has been implemented and runs on the Texas Instruments OMAP1610 H2 mobile device development platform [48] and the QEMU [7] emulated Integrator/CP platform [2]. In this Section we evaluate the CuriOS implementation in terms of error recovery capabilities as well as performance and memory overheads. We also present a brief analysis of the refactoring effort involved in constructing CuriOS services.

6.1 Error Recovery

In order to evaluate the error recovery capabilities of the CuriOS implementation, we resort to fault injection experiments using a modified version of the QEMU emulator. We have verified that error recovery works equally well on real hardware and we only use the emulator in order to enable non-intrusive and large-scale automated fault injection. Our QEMU-based fault injection tool picks a random instruction in pre-specified functions and injects a fault just before that instruction is executed. In each experiment run, we inject exactly one fault and observe the behavior of the system.

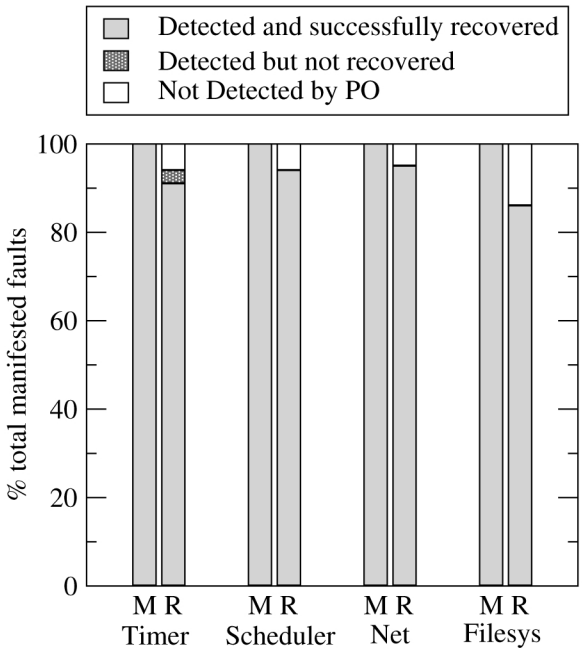


Figure 5: Error Recovery after Fault Injection

Our fault injection tool is used to inject two types of faults. The first type of fault is a memory access fault (an ARM processor "data abort"). These virtual memory faults are instantaneously detected at the injected instruction and immediately cause an error. The fault latency is zero in this case and error propagation is limited. The other type of fault that we inject is a register bit-flip. The tool randomly flips a bit in one of the register operands for the selected instruction. Register bit-flips do not always lead to errors (a corrupted register can be overwritten). They can, however, lead to latent errors which may not be detected immediately. This has been used to emulate several common programming errors such as incorrect assignment statements and pointer corruptions [38]. After recovery, we terminate experiments once we are able to verify that the OS is still functional (ability to schedule processes and access the disk).

We present error recovery results for the timer manager, system scheduler, networking stack and file system. In all of our experiments, we consider the system to be usable and successfully recovered if, at the end of the experiment, CuriOS can schedule new processes and can access the disk. We perform between 250 and 600 fault injection experiments per server and the results are shown in figure 5. The "M" columns present results for the memory access fault experiments and the "R" columns present results for the register bit-flip experiments. The Y axis represents the percentage of faults that had some visible manifestation (errors). Note that

all of these manifested faults would result in a service or system failure in most existing operating systems, which do not implement restart recovery with state management as used in CuriOS. An error is reported as successfully recovered if the system is usable after recovery. In some cases, a client's connection to a server is terminated because of a corrupted SSR or repeated errors while the system, as well as other clients using the restarted service, still remain usable. We count such cases as successful system recovery in the figure shown. In the face of arbitrary errors that corrupt a client's SSR or request, there is no possibility of maintaining the unfortunate client's connection.

Timer Manager: This experiment is set up so that a couple of processes that use the timer are started before faults are injected into the server. These are applications like a clock that displays the time of day. While the memory abort errors are fully recovered by reconstructing the internal timer queue, in the register bit-flip injections we see a few cases (3%) where errors are detected but are not recovered. These happen when the recovery procedure itself encounters an error and is unable to complete. We are working on eliminating such cases by improving the robustness of the recovery routines. 6% of the register-bit flips evade detection by the protected object mechanism and cause unrecoverable errors in other parts of CuriOS. This is because register bit-flip errors can propagate to other CuriOS subsystems through invalid method call arguments and results. We do not yet perform an exhaustive check of the validity of all method call arguments and results. This is a work in progress and we expect it to significantly reduce this inter-component error propagation.

System Scheduler: This experiment is set up similar to the timer manager experiment with several processes in the system. The goal of this experiment is to examine if a failure in the scheduler can be recovered and if CuiK can continue scheduling processes. For the memory abort experiments, re-creation of the internal linked list is always successful. In the case of the register bit-flip experiments, 6% of the errors are not detected by the PO mechanism and cause CuriOS to crash.

Network: We run a simple web server and an echo server in CuriOS while also running an HTTP client that fetches a half-megabyte file from an external host. Faults are injected into the LWIP code for TCP processing of IP packets on both the send and receive paths. If an error is detected, the TCP stack PO is restarted and the request is retried. If multiple attempts at executing a protected method fail, an exception is thrown. If this exception is thrown when processing an incoming IP packet, the packet is silently dropped by the IP layer. This has no effect on the correctness of TCP because this is similar to packet loss on the network and is recovered by the TCP stack. If an exception is thrown back to a client with a TCP connection handle, the TCP connection for that client is terminated. We verify that the network stack is still usable and other connections are unaffected in spite of a single connection failure. For the network stack, 5% of the manifested register-bit flip faults are not detected and consequently, not recovered. While the system is recovered in 95% of the cases, 45% of these recoveries were at the cost of a single client's TCP connection termination due to state corruption.

File System: We inject faults into the code for the ext2 file system that is used when accessing a file on disk (in `Ext2Inode`). The memory abort faults are always completely recovered after a retry. 13% of the manifested register bit-flip faults are not detected by the protected object mechanism and are therefore not recovered. Again, this is due to error propagation via corrupted method call arguments and results.

6.2 Performance

A protected method call incurs additional processing overhead in comparison to a normal C++ method call. We made use of both of CuriOS' supported platforms to measure the overhead associated with protected method calls. On the OMAP1610 hardware platform we measured the overhead in terms of microseconds of execution, and on the QEMU emulator we measured the overhead in terms of instructions executed. The OMAP1610 was clocked at 96MHz, and the same test source code was used for both platforms. Table 2 shows the overheads for the two types of protected method calls: a protected call into a stateless server that does not require the mapping of an SSR and a protected call info a server that uses an SSR to manage state information. In the second case, additional processing is required to map the SSR into memory. The time overhead for switching into and out of a protected object domain is comparable to the cost of performing two context switches (148 microseconds for two switches) in CuriOS. Since a protected method call is analogous to switching between two microkernel domains, we believe that this represents acceptable performance. The numbers reported here are the average

Table 2: Protected Method Call Performance

Protected Call	**Instruction Overhead**	**Time Overhead (microseconds)**
Without SSR	1594 ± 4	195.7 ± 0.5
With SSR	4893 ± 3	378.9 ± 0.9

of 100 trials with error estimates provided by the sample standard deviation. We believe that these overheads may be further reduced with careful code optimization.

Apart from the extra code implementing the protected object mechanism, a major source of overhead is the need to flush the TLB when switching between page tables. While the ARM architecture allows for selective flushing of TLB entries, our current implementation does not support this feature. The single address space design of CuriOS helps to keep the costs of protected method calls down by obviating the need to flush the virtually tagged caches on the OMAP1610 ARM processor.

How fast does recovery happen? When an error is detected, the exception handling framework signals the error and the C++ library unwinds the stack and destroys stack objects. Restarting the server requires re-running the constructor for the PO and code to recover information from SSRs (if required). Altogether, the time from error detection to a recovered system is usually on the order of a few hundred microseconds.

6.3 Memory Overheads

Protected objects, like user applications, require additional page tables to enforce memory protection and this results in some memory overhead. Each PO also has an associated heap and a stack for each thread that can execute within the protected domain. The memory overhead due to stacks depends on the number of threads that use the PO. The use of SSRs also results in some memory overheads. We use hardware protection to isolate SSRs. However, hardware protection is not always available for small memory regions. Thus the minimum size of an SSR is determined by the smallest hardware-protectable region of memory. For example, on the ARM platform, this is a 1 KB page. Our current implementation uses a page for the minimum size of an SSR. This results in some memory waste. If this is a concern for small embedded devices, our design can be extended so that multiple SSRs share the same protected area. This saves space at the cost of better isolation between the SSRs. This problem may be mitigated by future architectural support for finer granularity of access control such as Mondriaan Memory Protection [54]. Nevertheless, the total memory overhead per protected object in CuriOS is only on the order of tens of kilobytes when there are a small number of clients. This includes 20 KB for a minimal set of page tables plus memory pages for the heap, per-thread stack and per-client SSR (at least one page for each).

6.4 Refactoring Effort

Our proposed OS design requires writing OS service code to encapsulate objects in protected domains and to utilize our state management framework. The protected object support in CuriOS is implemented through wrapper objects. Wrappers are currently written by hand and consist of a one line statement per object method. The statement is a C++ preprocessor macro that expands to the code required to switch into and out of the associated protection domain. This additional complexity may also be avoided by using an automated wrapper generation tool. Code-changes are also required to refactor OS services so that they can make use of the state management framework. The use of SSR-based state management in the file system, scheduler and the timer manager required less than 50 additional lines of code in each component. In order to convert the LWIP networking stack to use SSRs, we had to change around 100 lines of code. This mostly involved replacing calls to its internal allocator with the SSR-based state management code.

7 Discussion

7.1 Security

Our security model relies primarily on address space isolation. We only map in memory that is necessary for a protected object to execute. This includes the unprivileged code and stack for the object as well as the SSR region for the request. Our model is most closely related to Nooks, which uses similar protection policies for kernel memory. We differ from Nooks in that protected objects execute in an unprivileged processor mode. This prevents a malfunctioning or compromised server from affecting the integrity or confidentiality of information used by inactive clients. Although we restrict the scope of possible damage, our current implementation does not consider intentionally malicious modules. We are working on fortifying the protected method call and server state management mechanisms by borrowing ideas from systems like EROS.

7.2 Fault-Tolerance

A number of standard fault tolerance techniques are available in literature. These include redundancy in hardware and software, transactions, error correction codes for memory, majority or Byzantine voting, and other software fault tolerance approaches [49]. Some of these techniques can be directly applied to CuriOS to further improve its fault tolerance. These techniques may be used to ensure that the core of the system (CuiK and recovery code) itself is protected from failure.

VINO [42] used transactions to roll-back changes made by misbehaving kernel extensions. We have also investigated the use of software transactional memory techniques to protect component state in Choices [15].

The use of transactional semantics alone to recover complete component state is only effective when errors are detected before commits. When this property cannot be enforced, there are no constraints on error propagation within the component. However, when used together with our SSR-based approach that reduces error propagation, transactions can provide an additional layer of protection to SSRs while they are being manipulated by a service.

In addition to some of the operating systems discussed in Section 2, many other system designs incorporate virtual memory protection to improve reliability. In the Rio project [12], virtual memory was used to protect the file cache from corruption by errors occurring elsewhere in the system. The protected object concept is similar to a virtual memory protected region in Nooks [46]. However, unlike Nooks, a protected object executes in an unprivileged processor mode. More importantly, while Nooks is designed to wrap OS extensions such as device drivers, a protected object can encapsulate core OS components. Unlike the shadow driver mechanism [45] used by Nooks, the SSR-based recovery mechanisms can isolate requests that cause crashes because of a software bug and continue servicing requests that do not trigger the bug. This is possible because of the rigorous partitioning of per-client state in CuriOS. When using the shadow driver approach, the bug will be triggered in the shadow driver just as it was in the original driver since the same code is used.

OS service design using SSRs is closely related to the principle of crash-only software [9]. Similar to crash-only components, recovery involves a component restart and component crashes are masked from end users using transparent component-level retries.

7.3 Applicability to Other Systems

The state separation approach described in this work may also be applied to other microkernel systems which provide isolation for OS services such as L4 and Minix3. This would require some modifications to these kernels to incorporate SSR management and changes to server APIs. These systems would need to also be augmented to support the other requirements for transparent recovery detailed in Section 3. The benefits of state partitioning for operating systems that do not use inter-component isolation is debatable. Since there are no constraints on error propagation, it is difficult to determine which OS subsystem needs to be restarted.

7.4 Additional Benefits

There are several additional benefits of our design. Since memory usage of SSRs can be attributed to clients, they cannot cause a denial of service problem at a server by creating a large number of connections to it. Our design also makes it possible to transparently upgrade a server by simply terminating the old server and starting a newer version while preserving the SSRs. If the new server can interpret the existing SSRs (backwards compatible), it can continue serving existing clients.

7.5 Drawbacks

While there are several advantages of adopting our approach to OS design, there are also several drawbacks. Apart from the performance and memory overheads quantified in Section 6, there is still the added complexity involved in separating state from services and hopefully not introducing new software faults (bugs) in the process. We have tried to quantify this additional complexity in terms of lines of code in Section 6. Our observations indicate that it requires about 12-24 person-hours to design and refactor an OS service to work with our framework. This includes the time spent in fixing most bugs uncovered using fault injection.

8 Concluding Remarks

In this paper, we have analyzed some of the reasons why current designs for reliable microkernel operating systems struggle with client-transparent recovery. Through simple fault injection experiments with various systems, we gain insights into properties that are essential for successful client-transparent recovery of OS services. We have described a design for structuring an OS that preserves these properties. CuriOS minimizes error propagation and persists client information using distributed and isolated OS service state to enhance the transparent restartability of several system components. Restricted memory access permissions prevent erroneous OS services from corrupting arbitrary memory locations. Our experimental results show that it is possible to isolate and recover core OS services from a significant percentage of errors with acceptable performance.

The source code for our CuriOS implementation and the code for the QEMU-based fault injector can be found on our website at `http://choices.cs.uiuc.edu/`.

Acknowledgments

We are very grateful for the insights and feedback from Galen Hunt (our shepherd) and the anonymous reviewers. Part of this research was made possible by grants from DoCoMo Labs USA and Motorola as well as generous equipment support from Texas Instruments.

References

[1] ABROSSIMOV, V., AND HEMANN, F. Fast Error Recovery in CHORUS/OS: The Hot-Restart Technology. Tech. Rep. CSI-T4-96-34, Chorus Systems, Inc., August 1996.

[2] ARM™Integrator Family. `http://www.arm.com/products/DevTools/Hardware_Platforms.html`.

[3] AVIZIENIS, A., LAPRIE, J.-C., RANDELL, B., AND LANDWEHR, C. E. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing 1*, 1 (2004), 11–33.

[4] BARTLETT, J., GRAY, J., AND HORST, B. Fault Tolerance in Tandem Computer Systems. In *The Evolution of Fault-Tolerant Systems*, A. Avizienis, H. Kopetz, and J.-C. Laprie, Eds. Springer-Verlag, Vienna, Austria, 1987, pp. 55–76.

[5] BARTLETT, J. F. A NonStop Kernel. In *Symposium on Operating Systems Principles* (New York, NY, USA, 1981), ACM Press, pp. 22–29.

[6] BARTLETT, W., AND SPAINHOWER, L. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing 1*, 1 (2004), 87–96.

[7] BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track* (April 2005).

[8] CAMPBELL, R. H., JOHNSTON, G. M., AND RUSSO, V. "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)". *ACM Operating Systems Review 21*, 3 (July 1987), 9–17.

[9] CANDEA, G., AND FOX, A. Crash-Only Software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)* (Lihue, HI, May 2003).

[10] CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. Microreboot – A Technique for Cheap Recovery. In *Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 31–44.

[11] CHASE, J. S., LEVY, H. M., FEELEY, M. J., AND LAZOWSKA, E. D. Sharing and Protection in a Single Address Space Operating System. *ACM Transactions on Computer Systems 12*, 4 (1994), 271–307.

[12] CHEN, P. M., NG, W. T., CHANDRA, S., AYCOCK, C., RAJAMANI, G., AND LOWELL, D. The Rio File Cache: Surviving Operating System Crashes. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (1996), pp. 74–83.

[13] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. R. An Empirical Study of Operating System Errors. In *Symposium on Operating Systems Principles* (2001), pp. 73–88.

[14] Compressed ROM filesystem. `http://sourceforge.net/projects/cramfs/`.

[15] DAVID, F. M., AND CAMPBELL, R. H. Building a Self-Healing Operating System. In *Symposium on Dependable, Autonomic and Secure Computing* (Columbia, MD, Sep 2007), pp. 3–17.

[16] DAVID, F. M., CARLYLE, J. C., AND CAMPBELL, R. H. Exploring Recovery from Operating System Lockups. In *USENIX Annual Technical Conference* (Santa Clara, CA, June 2007), pp. 351–356.

[17] DAVID, F. M., CARLYLE, J. C., CHAN, E. M., RAILA, D. K., AND CAMPBELL, R. H. *Exception Handling in the Choices Operating System*, vol. 4119 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 2006.

[18] DAVID, F. M., CARLYLE, J. C., CHAN, E. M., REAMES, P. A., AND CAMPBELL, R. H. Improving Dependability by Revisiting Operating System Design. In *Workshop on Hot Topics in Dependability* (Edinburgh, UK, June 2007), pp. 58–63.

[19] DENNING, P. J. Fault Tolerant Operating Systems. *ACM Computing Survey 8*, 4 (1976), 359–389.

[20] DUNKELS, A. Full TCP/IP for 8-bit Architectures. In *International Conference on Mobile Systems, Applications and Services* (New York, NY, USA, 2003), ACM, pp. 85–98.

[21] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, J. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Symposium on Operating Systems Principles* (1995), pp. 251–266.

[22] GU, W., KALBARCZYK, Z., AND IYER, R. K. Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors. In *International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 887–896.

[23] HARTIG, H., HOHMUTH, M., AND WOLTER, J. Taming Linux. In *Australasian Conference on Parallel And Real-Time Systems* (Adelaide, Australia, Sept 1998).

[24] HERDER, J., BOS, H., GRAS, B., HOMBURG, P., AND TANENBAUM, A. S. Roadmap to a Failure-Resilient Operating System. *USENIX ;login 32* (February 2007), 14–20.

[25] HERDER, J. N., BOS, H., GRAS, B., HOMBURG, P., AND TANENBAUM, A. S. Reorganizing UNIX for Reliability. In *Asia-Pacific Computer Systems Architecture Conference* (2006), pp. 81–94.

[26] HERDER, J. N., BOS, H., GRAS, B., HOMBURG, P., AND TANENBAUM, A. S. Failure Resilience for Device Drivers. In *International Conference on Dependable Systems and Networks* (2007), pp. 41–50.

[27] HILLER, M., JHUMKA, A., AND SURI, N. PROPANE: An Environment for Examining the Propagation of Errors in Software. In *Symposium on Software Testing and Analysis* (New York, NY, USA, 2002), ACM Press, pp. 81–85.

[28] HUNT, G. C., LARUS, J. R., ABADI, M., AIKEN, M., BARHAM, P., FAHNDRICH, M., HAWBLITZEL, C., HODSON, O., LEVI, S., MURPHY, N., STEENSGAARD,

B., TARDITI, D., WOBBER, T., AND ZILL, B. An Overview of the Singularity Project. Tech. Rep. MSR-TR-2005-135, Microsoft Research, 2005.

[29] HYDEN, E. A. *Operating System Support for Quality of Service*. PhD thesis, University of Cambridge, 1994.

[30] JOHANSSON, R. On Single Event Upset Error Manifestation. In *European Dependable Computing Conference* (London, UK, 1994), Springer-Verlag, pp. 217–231.

[31] KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., NO, H. M. B., HUNT, R., MAZIÈRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. Application Performance and Flexibility on Exokernel Systems. In *Symposium on Operating Systems Principles* (New York, NY, USA, 1997), ACM Press, pp. 52–65.

[32] KENNELL, R., AND JAMIESON, L. H. Establishing the Genuinity of Remote Computer Systems. In *USENIX Security Symposium* (2003), pp. 295–308.

[33] KITTUR, S., ARMAND, F., STEEL, D., AND LIPKIS, J. Fault Tolerance in a Distributed CHORUS/MiX System. In *USENIX Annual Technical Conference* (1996), pp. 219–228.

[34] LEE, I., AND IYER, R. K. Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System. In *International Symposium on Fault-Tolerant Computing* (1993), pp. 20–29.

[35] LESLIE, B., VAN SCHAIK, C., AND HEISER, G. Wombat: A portable user-mode Linux for embedded systems. In *Linux.Conf.Au, (Canberra)* (April 2005).

[36] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GÖTZ, S. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Symposium on Operating Systems Design and Implementation* (San Francisco, CA, Dec. 2004), pp. 17–30.

[37] LIEDTKE, J. On μ-Kernel Construction. In *Symposium on Operating Systems Principles* (New York, NY, USA, 1995), ACM Press, pp. 237–250.

[38] NG, W. T., AND CHEN, P. M. The Systematic Improvement of Fault Tolerance in the Rio File Cache. In *International Symposium on Fault-Tolerant Computing* (1999), pp. 76–83.

[39] RANDELL, B. Operating Systems: The Problems of Performance and Reliability. In *IFIP Congress 71 Volume 1* (1971), pp. 281–290.

[40] ROZIER, M., ABROSSIMOV, V., ARMAND, F., BOULE, I., GIEN, M., GUILLEMONT, M., HERRMAN, F., KAISER, C., LANGLOIS, S., LÉONARD, P., AND NEUHAUSER, W. Overview of the Chorus Distributed Operating System. In *Workshop on Micro-Kernels and Other Kernel Architectures* (Seattle WA (USA), 1992), pp. 39–70.

[41] SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. Design and Implementation of the Sun Network Filesystem. In *USENIX Conference* (Portland, OR, USA, 1985), pp. 119–130.

[42] SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Symposium on Operating Systems Design and Implementation* (New York, NY, USA, 1996), ACM, pp. 213–227.

[43] SHAPIRO, J. S. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, 1999.

[44] The Standish Group. TCO in the Trenches 2002. `http://www.himalaya.compaq.com/object/TCO.html`.

[45] SWIFT, M. M., ANNAMALAI, M., BERSHAD, B. N., AND LEVY, H. M. Recovering Device Drivers. In *Symposium on Operating Systems Design and Implementation* (2004), pp. 1–16.

[46] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the Reliability of Commodity Operating Systems. In *Symposium on Operating Systems Principles* (New York, NY, USA, 2003), ACM Press, pp. 207–222.

[47] TANENBAUM, A. S., HERDER, J. N., AND BOS, H. Can We Make Operating Systems Reliable and Secure? *IEEE Computer 39*, 5 (2006), 44–51.

[48] Texas Instruments OMAP Platform. `http://focus.ti.com/omap/docs/omaphomepage.tsp`.

[49] TORRES-POMALES, W. Software Fault Tolerance: A Tutorial. Tech. Rep. NASA/TM-2000-210616, NASA Langley Research Center, 2000.

[50] ÚLFAR ERLINGSSON, VALLEY, S., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: Software Guards for System Address Spaces. In *Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 75–88.

[51] VENEMA, W. Murphy's law and computer security. *USENIX Security Symposium* (1996), 187.

[52] WANG, L., KALBARCZYK, Z., GU, W., AND IYER, R. K. An OS-level Framework for Providing Application-Aware Reliability. In *IEEE Pacific Rim International Symposium on Dependable Computing* (2006).

[53] WELCH, B. B. *Naming, State Management, and User-Level Extensions in the Sprite Distributed File System*. PhD thesis, University of California, Berkeley, CA 94720, Feb. 1990. Technical Report UCB/CSD 90/567.

[54] WITCHEL, E., CATES, J., AND ASANOVIĆ, K. Mondrian Memory Protection. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2002), ACM Press, pp. 304–316.

[55] ZHOU, F., CONDIT, J., ANDERSON, Z., BAGRAK, I., ENNALS, R., HARRE, M., NECULA, G., AND BREWER, E. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Symposium on Operating Systems Design and Implementation* (Nov 2006), pp. 45–60.

Redline: First Class Support for Interactivity in Commodity Operating Systems

Ting Yang *tingy@cs.umass.edu*
Tongping Liu *tonyliu@cs.umass.edu*
Emery D. Berger *emery@cs.umass.edu*
Scott F. Kaplan† *sfkaplan@cs.amherst.edu*
J. Eliot B. Moss *moss@cs.umass.edu*

Dept. of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003-9264

†*Dept. of Mathematics and Computer Science*
Amherst College
Amherst, MA 01002-5000

Abstract

While modern workloads are increasingly interactive and resource-intensive (e.g., graphical user interfaces, browsers, and multimedia players), current operating systems have not kept up. These operating systems, which evolved from core designs that date to the 1970s and 1980s, provide good support for batch and command-line applications, but their *ad hoc* attempts to handle interactive workloads are poor. Their best-effort, priority-based schedulers provide no bounds on delays, and their resource managers (e.g., memory managers and disk I/O schedulers) are mostly oblivious to response time requirements. Pressure on any one of these resources can significantly degrade application responsiveness.

We present Redline, a system that brings first-class support for interactive applications to commodity operating systems. Redline works with unaltered applications and standard APIs. It uses lightweight specifications to orchestrate memory and disk I/O management so that they serve the needs of interactive applications. Unlike real-time systems that treat specifications as strict requirements and thus pessimistically limit system utilization, Redline dynamically adapts to recent load, maximizing responsiveness and system utilization. We show that Redline delivers responsiveness to interactive applications even in the face of extreme workloads including fork bombs, memory bombs and bursty, large disk I/O requests, reducing application pauses by up to two orders of magnitude.

1 Introduction

The enormous increases in processing power, memory size, storage capacity, and network bandwidth over the past two decades have led to a dramatic change in the richness of desktop environments. Users routinely run highly-graphical user interfaces with resource-intensive applications ranging from video players and photo editors to web browsers, complete with embedded Javascript and Flash applets.

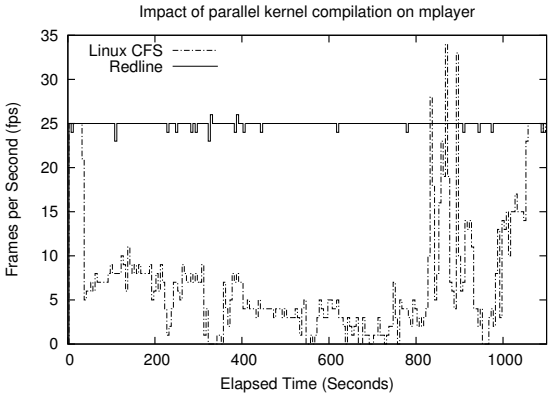


Figure 1: The frame rate of mplayer when performing a Linux kernel compiling using make -j32 on a standard Linux 2.6 kernel and on the Redline kernel.

Unfortunately, existing general-purpose operating systems do not provide adequate support for these modern applications. Current operating systems like Windows, Linux, and Solaris were designed using standard resource management policies and algorithms, which in turn were not developed with interactivity in mind. While their CPU schedulers attempt to enhance interactive behavior, their memory managers and I/O managers focus on increasing system throughput rather than reducing latency. This lack of coordination between subsystems, and the fact that they can work at cross purposes, means that pressure on any resource can significantly degrade application responsiveness.

For example, a memory-intensive application can cause the system to evict pages from the graphical user interface, making the system as a whole unresponsive. Similarly, disk-intensive applications can easily saturate I/O bandwidth, making applications like video players unusable. Furthermore, while best-effort, priority-based schedulers are a good match for batch-style applications, they provide limited support for ensuring responsiveness. Activities that

strain the resources of a system—image or video processing, working with large data files or file systems, or switching frequently between a number of active applications—are likely to cause one of the resource managers to underallocate resources to some interactive tasks, making those tasks respond poorly.

As an example, Figure 1 shows the frame rate of mplayer, a movie player, while a Linux kernel compilation, invoked using make -j32, is performed using both a standard Linux 2.6.x kernel and our Redline kernel. For the standard kernel, the compilation substantially degrades the interactivity of the movie player, rendering the video unwatchable. Worse, the entire GUI becomes unresponsive. Similar behavior occurs on Windows when watching a video using the Windows Media Player while running a virus scan in the background.

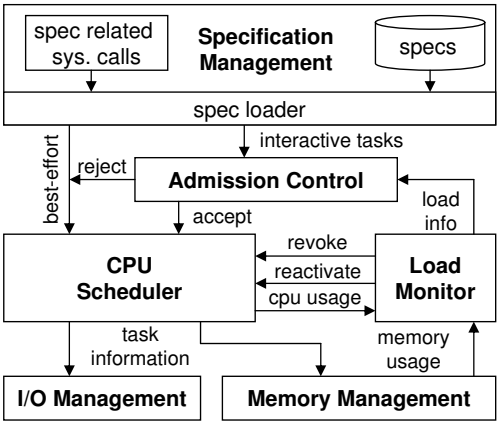


Figure 2: The Redline system.

Contributions

We present Redline, a system that integrates resource management (memory management and disk I/O scheduling) with the CPU scheduler, orchestrating these resource managers to maximize the responsiveness of interactive applications.

Redline relies on *lightweight specifications* to provide enough information about the response time requirements of interactive applications. These specifications, which extend Rialto's *CPU specifications* [15], give an estimate of the resources required by an application over any period in which they are active. These specifications are concise, consisting of just a few parameters (Section 3), and are straightforward to generate: a graduate student was able to write specifications for a suite of about 100 applications—including Linux's latency-sensitive daemons—in just one day. In an actual deployment, we expect these specifications to be provided by application developers.

Each resource manager uses the specifications to inform its decisions. Redline extends a standard time-sharing CPU scheduler with an *earliest deadline first (EDF)*-based scheduler [18] that uses these specifications to schedule interactive applications (Section 6). Redline's memory manager protects the working sets of interactive applications, preferentially evicting pages from non-interactive applications and further reducing the risk of paging by maintaining a pool of free pages available only to interactive tasks—a *rate-controlled memory reserve* (Section 4). Redline's disk I/O manager avoids pauses in interactive applications by dynamically prioritizing these tasks according to their need to complete the I/O operation and resume execution on the CPU (Section 5).

Unlike real time systems that sacrifice system utilization in exchange for strict guarantees [15, 16], Redline provides strong isolation for interactive applications while ensuring high system utilization. Furthermore, Redline works with standard APIs and does not require any alterations to existing applications. This combination makes Redline practical for use in commodity operating system environments.

We have implemented Redline as an extension to the Linux kernel (Section 7). We present the results of an extensive empirical evaluation comparing Redline with the standard Linux kernel (Section 8). These results demonstrate Redline's effectiveness in ensuring the responsiveness of interactive applications even in the face of extreme workloads, including bursty I/O-heavy background processes, fork bombs, and memory bombs.

2 Redline Overview

This paper focuses on Redline's support for interactive and best-effort tasks:

1. **Interactive (Iact)**: response-time sensitive tasks that provide services in response to external requests or events. These include not only tasks that interact with users, but also tasks that serve requests from other tasks, such as kernel daemons;

2. **Best-effort (BE)**: tasks whose performance is not critical, such as virus scanners.

Figure 2 presents an overview of the Redline system. At the top, the specification management allows a system administrator to provide specifications for a set of important applications. Redline loads each specification from a file whenever the corresponding application is launched. Redline treats any application without a specification as a best-effort task.

Whenever a new task is launched, Redline performs admission control to determine whether the system can accommodate it. Specifically, Redline dynamically tracks the load consumed by the active interactive tasks, and then uses the new task's specification to determine whether the resources are available to support it as an interactive task.

Once the admission control mechanism accepts a task, Redline propagates its specification to the memory manager, the disk I/O manager, and the CPU scheduler. The memory manager uses the specification to protect the task's working set, preferentially evicting either non-working-set pages or pages from best-effort tasks. It also maintains a specially controlled pool of free pages for interactive tasks, thus isolating them from best-effort tasks that allocate aggressively. The disk I/O management assigns higher priorities to interactive tasks, ensuring that I/O requests from interactive tasks finish as quickly as possible. Finally, Redline's extended CPU scheduler provides the required CPU resources for the interactive tasks.

Unlike real-time systems, which pessimistically reject jobs if the combined specifications would exceed system capacity, Redline optimistically accepts new jobs based on the actual usage of the system, and adapts to overload if necessary. When Redline detects an overload—that the interactive tasks are trying to consume more resources than are available—then the load monitor selects a victim to downgrade, making it a best-effort task. This victim is the task that acts least like an interactive task (i.e., it is the most CPU-intensive). This strategy allows other interactive tasks to continue to meet their response-time requirements. Whenever more resources become available, Redline will promote the downgraded task, again making it interactive.

By integrating resource management with appropriate admission and load control, Redline effectively maintains interactive responsiveness even under heavy resource contention.

3 Redline Specifications

Under Redline, each interactive application (e.g., a text editor, a movie player, or a web browser) must have a specification to ensure its responsiveness. However, the application itself depends on other tasks that must also be responsive. Specifically, the graphical user interface (e.g., the X Windows server, the window/desktop manager) comprise a set of tasks that must be given specifications to ensure that the user interface remains responsive. Similarly, specifications for a range of kernel threads and daemons allow Redline to ensure that the system remains stable. Finally, specifications for a range of administrative tools (e.g., bash, top, ls, and kill) make it possible to manage the system even under extreme load.

A specification in Redline is an extension of *CPU reservations* [15] that allow an interactive task to reserve C milliseconds of computation time out of every T milliseconds. A specification consists of the following fields:

$$\langle pathname{:}type{:}C{:}T{:}flags{:}\pi{:}io \rangle$$

The first field is the pathname to the executable, and the second field is its type (usually lact). Two important flags include I, which determines whether the specification can be inherited by a child process, and R, which indicates if

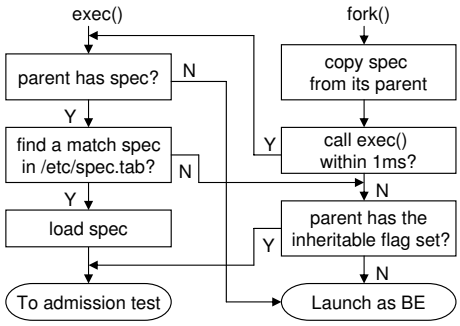


Figure 3: Loading specifications in Redline

the specification may be revoked when the system is overloaded. A specification can also contain two optional fields: π is the memory protection period in seconds (see Section 4) and *io* is the I/O priority (see Section 5).

For example, here is the specification for *mplayer*, an interactive movie player:

$$\langle \text{/usr/bin/mplayer:lact:5:30:IR:-:-} \rangle$$

This specification indicates that *mplayer* is an interactive task that reserves 5 ms out of each 30 ms period, whose specification is inheritable, that its interactive status can be revoked if necessary, and whose memory protection period and I/O priority are chosen by Redline automatically.

Setting specifications: We derived specifications for a range of applications by following several simple rules. Because most administrative tools are short-lived, reserving a small percentage of the CPU bandwidth over hundreds of milliseconds is sufficient to ensure responsiveness. While most kernel threads and daemons are not CPU intensive, they are response-time sensitive, so their reservation period should be in the tens of milliseconds. Finally, for interactive applications like a movie player, the reservation period should be around 30 ms to ensure 30 frames per second, which implies that the X server and window/desktop manager should also use the same reservation period.

We found that setting specifications was straightforward. It took one graduate student a single work day to manually generate a set of specifications for about 100 applications in a Linux system using the K Desktop Environment (KDE). Because this specification file is portable, specifications for a wide variety of applications could easily be shipped with an operating system distribution.

Loading specifications: Redline stores its specifications in a file (/etc/spec/spec.tab). An interactive task either uses the specification loaded from this file when exec() is invoked, or it adopts the one inherited from its parent. Figure 3 shows how Redline loads the specification for each task.

In practice, most tasks that invoke exec() do so shortly after being forked. Therefore, when a new task is forked, Redline gives it a 1 ms execution window to perform an

exec(). If it does so, and if the parent task is itself an interactive task, then Redline searches the specification file for an entry with a matching path name. If a match is found, the specification provided is adopted for this new task.

If there is no entry for the application, or if the task does not invoke exec() during that initial 1 ms window, then Redline examines whether the task should inherit its parent task's specification. If that specification is marked as *inheritable*, then it is applied to the new task. Under all other circumstances, the new task is classified as best-effort.

If the new task's specification classify it as interactive, then Redline will submit the task to admission control. If Redline determines that the load is low enough to support the resource needs of this new interactive task, then it is admitted; otherwise, it is demoted to be a best-effort task.

Note that in the absence of any specifications, all tasks become best-effort tasks, and Redline acts like a standard system without special support for interactivity.

4 Redline Virtual Memory Management

The goal of existing virtual memory managers (VMM) in commodity operating systems is to maximize overall system throughput. Most VMM's employ "use-it-or-lose-it" policies under which memory referencing speed determines allocation: the more pages a task references per second, the larger its main memory allocation.

A task that blocks on I/O is more vulnerable to losing its allocation and then later being forced to page-swap when it awakens. Because typical VMM's do not isolate each task's allocation, a single memory-intensive task can "steal" the allocations of other tasks that are not actively using their pages, causing the other tasks to page-swap more often. Worse, page swapping itself is the kind of blocking I/O operation that can cause a task to lose more of its allocation, exacerbating the problem.

Because interactive tasks routinely block on user input, they are especially susceptible to allocation loss. If an interactive task has an insufficient allocation, its execution is likely to be interrupted by lengthy page-swap operations, leaving it unresponsive, and making it even more susceptible to further allocation loss.

To address this problem, the Redline VMM is designed to keep interactive tasks responsive even after blocking on input. Redline uses three mechanisms to achieve these goals: *protecting the working sets* of interactive tasks, using a *rate-controlled memory reserve*, and setting *speed-bump* pages.

Protecting working sets: The Linux VMM uses a page replacement algorithm that approximates a *global least recently used (gLRU)* policy. Pages are approximately ordered by recency of use, without consideration of the task to which each page belongs. The VMM cleans and selects for reclamation the least-recently used pages. The implicit goal of this policy is to minimize the total number of page swaps performed by the system, irrespective of how any one process performs.

For an interactive task to remain responsive, a VMM must keep its *working set*—those pages that are currently in active use—resident in memory. Under gLRU, a portion of a task's working set may be evicted from main memory if the system-wide demand for main memory is large enough, and if a task does not reference its pages rapidly enough. Under this memory pressure, interactive tasks can quickly become non-responsive.

Under the Redline VMM, each interactive task can specify a *memory protection period* π. The VMM will evict a page only if it has not been referenced for at least π seconds—that is, if the page has *expired*. By default, $\pi = 30 \times 60$, or 30 minutes if π is not supplied by the specification. This default period requires that a user ignore an interactive application for a substantial period of time before the VMM can evict its working set.

The Redline VMM handles the pages of interactive tasks and best-effort tasks differently. If page reclamation is caused by a best-effort task, then the VMM reclaims pages using the system's default VMM mechanism, but with a slight modification: only pages belonging to best-effort tasks and expired pages belonging to interactive tasks may be reclaimed. Among these two types of pages eligible for reclamation, the VMM selects the least recently used pages.

However, if page reclamation is caused by an interactive task, then the VMM first tries to use the *rate-controlled memory reserve* (described in more detail below). If this reserve provides insufficient space, reclamation proceeds as above for best-effort tasks, reclaiming not recently used pages from best-effort tasks and expired pages from interactive tasks. If this attempt at reclamation is also insufficient, the VMM then reclaims more recently used pages from best-effort tasks. If even this aggressive approach is insufficient, then there is not enough memory to cache the working sets of all of the interactive tasks. In this case, the VMM demotes some interactive tasks to best-effort tasks. After taking this step, the VMM repeats the above steps, attempting to reclaim sufficient main memory space.

Rate-controlled reserve: The Linux VMM forces the allocating task to reclaim pages when the amount of free memory falls below a threshold. This approach is not desirable for interactive tasks because, should such a task fault, it would block during the potentially lengthy reclamation process. Instead, the Redline VMM maintains a small free-memory reserve (about 8 MB in our implementation) for interactive tasks, and controls the rate at which this reserve is consumed. Although an interactive task still may block during reclamation, the reserve makes that situation significantly less likely.

The Redline VMM gives each interactive task a reserve budget b (the default is 256 pages) and records the time t_f

when the first page in its budget is consumed. For each page consumed from the reserve, the Redline VMM reduces the budget of the consuming task and then triggers a kernel thread to reclaim pages in background if necessary. We do not want any one task to quickly exhaust the reserve and affect other tasks, so the rate at which a task consumes reserved pages should not be faster than the system can reclaim them. Therefore, the Redline VMM charges each reserved page a cost c that is roughly the overhead of one disk access operation (5 ms in Redline). When a task expends its budget, the VMM evaluates the inequality $t_f + bc < t$, where t is the current time. If the inequality is true, then the VMM adds b to the task's budget. If the inequality is false, the task is consuming reserved pages too quickly. Thus, the VMM prevents the task from using the reserve until b pages are reclaimed or the reserve resumes its full capacity. We show the effect of this limited isolation between interactive tasks Section 8.

Setting speed-bump pages: A VMM can only reclaim a *clean* (unmodified) page; *dirty* (modified) pages must be copied to the backing store, thus cleaning them, before they can be reclaimed. Sometimes, a best-effort task may dirty pages faster than the VMM can clean them, thus preventing the VMM from reclaiming those pages. If enough such pages are unreclaimable, the VMM may be unable to cache the working sets of interactive tasks, thus allowing one best-effort task to degrade interactivity.

To prevent best-effort tasks from "locking" memory in this manner, whenever the Redline VMM is aggressively searching for pages to reclaim (see above) and finds a page belonging to a best-effort task, it removes access permissions to that page and marks it as a *speed-bump page*. If the task then references that page, execution traps into the kernel and the VMM notices that the referenced page is a speed-bump. Before restoring access permissions and resuming execution, Redline suspends the task briefly (e.g., 100 ms). The VMM therefore slows the task's memory reference rate, giving the VMM enough time to reclaim more of its pages.

5 Redline Disk I/O Management

Like the VMM, the I/O manager of a general-purpose OS does not distinguish between interactive and best-effort tasks. The policies that determine when and in what order pages are read from and written to disk are designed to optimize system throughput and are oblivious to CPU scheduler goals. This obliviousness can lead the I/O manager to schedule the requests for best-effort tasks before those of interactive tasks in a way that substantially harms response times.

To prevent this problem, the Redline I/O manager manages key I/O events in both the file system and block device layer so that responsive tasks can meet their deadlines.

Journaling: For Linux, *ext3* is a journaling file system that is the default for most distributions. Like any journaling file system, *ext3* commits its updates as atomic transactions, each of which writes a group of cached, dirty pages along with their new metadata. Its implementation is designed to maximize system-wide throughput, sometimes to the detriment of CPU scheduling goals. We describe here a particular problem with this file system's implementation that Redline fixes. Although this particular problem is specific to Linux's *ext3*, it is representative of the way in which any OS component that manages system resources can undermine interactivity.

Consider two tasks: an interactive task P_i, and a best-effort task P_{be}, which simultaneously use the write() system call to save data to some file on the same *ext3* file system. These system calls will not immediately initiate disk activity. Instead, the data written via this mechanism will be buffered as a set of dirty pages in the file system cache. Critically, these pages will also be added to a single, global, *compound* transaction by *ext3*. This transaction will thus contain dirty pages from any file written by any task, including pages written by both P_i and P_{be}.

Consider the case that P_{be} writes a large amount of data through write(), while P_i writes a small amount. Furthermore, after both tasks have performed these write() operations, suppose that P_i performs an fsync() system call to ensure that its updates are committed to disk. Because of the compound transactions used by *ext3*, P_i will block until both its own dirty pages and those of P_{be} are written to disk.

If the OS caches too many pages written by P_{be}, then the fsync() operation will force P_i to become noticeably unresponsive. This poor interaction between compound transactions and fsync() occurs not only for *ext3*, but also for *ReiserFS*[25]. Under Linux, the *dirty threshold* d is a system-wide parameter that determines what percentage of main memory may hold dirty pages—pages that may belong to any task—before a disk transfer is initiated to "clean" those pages. By default, $d = 10\%$, making it possible on a typical system for 100 MB to 200 MB of dirty pages to be cached and then written synchronously when fsync() is called.

Redline takes a number of steps to limit this effect. First, Redline assigns different dirty thresholds for each type of task (RT:10%, Iact:5%, BE:2MB). Redline further restricts the threshold for best-effort tasks to a constant limit of 2 MB, ensuring that no matter the size of main memory, best-effort tasks cannot fill the compound transactions of some journaling file system with a large number of dirty pages. Finally, Redline assigns the kernel task that manages write operations for each file system (in Linux, kjournald) to be an interactive task, ensuring that time-critical transaction operations are not delayed by other demands on the system.

Block device layer: Much like the journaling file systems described above, a block device layer may have unified data structures, thresholds, or policies that are applied

irrespective of the tasks involved. These components are typically designed to maximize system-wide throughput. However, the unification performed by these components may harm the responsiveness of interactive tasks. Redline addresses these problems by handling these components of the block device layer on a per-task-type basis.

The Linux block device manager, dubbed *Completely Fair Queuing (CFQ)*, contains a single *request queue* that stores I/O requests from which actual disk operations are drawn. Although this request queue internally organizes I/O requests into classes (real-time, best-effort, and "idle"), it has a maximum capacity that is oblivious to these classes. When a new request is submitted, the *congestion control mechanism* examines only whether the request queue is full, with no consideration of the requesting task's class. If the queue is full, the submitting task blocks and is placed in a FIFO-ordered wait-queue. Thus, a best-effort task might rapidly submit a large number of requests, thus congesting the block device and arbitrarily delaying some interactive task that requests an I/O operation.

Redline addresses this problem by using multiple request queues, one per task class. If one of the queues fills, the congestion control mechanism will only block processes in the class associated with that queue. Therefore, no matter how many requests have been generated by best-effort tasks, those requests alone cannot cause an interactive task to block.

In addition, while the default request queue for CFQ is capable of differentiating between various request types, it does not provide sufficient isolation for interactive tasks. Specifically, once a request has been accepted into the request queue, it awaits selection by the I/O scheduler to be placed in the dispatch queue, where it is scheduled by a typical elevator algorithm to be performed by the disk itself. This default CFQ scheduler not only gives preference to requests based on their class, but also respects the priorities that tasks assign requests within each class. However, each buffered write request—the most common kind—is placed into a set of shared queues in best-effort class irrespective of the task that submitted the request. Therefore, best-effort tasks may still interfere with the buffered write requests of interactive tasks by submitting large numbers of buffered write requests.

Redline adds both an interactive lact class to the request queue management, matching its CPU scheduling classes. All write requests are placed into the appropriate request queue class based on the type of the submitting task. The I/O scheduler prefers requests from the interactive class over those in the BE class, thus ensuring isolation of the requests of interactive tasks from BE tasks.

Additionally, the specification for a task (see Section 3) includes the ability to specify the priority of the task's I/O requests. If the specification does not explicitly provide this information, then Redline automatically assigns a higher priority to tasks with smaller T values. Redline organizes requests on per-task basis within the given class, allowing the I/O scheduler provide some isolation between interactive tasks.

Finally, CFQ, by default, does not guard against starvation. A task that submits a low-priority I/O request into one of the lesser classes may never have that request serviced. Redline modifies the I/O scheduler to ensure that all requests are eventually served, preventing this starvation.

6 Redline CPU Scheduling

The time-sharing schedulers used by commodity operating systems to manage both interactive and best-effort tasks neither protect against overload nor provide consistent interactive performance. To address these limitations, Redline employs *admission control* to protect it against overload in most cases, and uses *load control* to recover quickly from sudden bursts of activity. Redline also uses an *Earliest Deadline First (EDF)*-based scheduler for interactive tasks to ensure that they receive CPU time as required by their specifications.

6.1 Admission and Load Control

Admission control determines whether a system has sufficient free resources to support a new task. It is required by any system that provides response time guarantees, such as real-time systems, though it is typically absent from commodity operating systems.

In real-time systems, admission control works by conservatively assuming that each task will always consume all of resources indicated in its specification. If the addition of a new task would cause the sum of the specified CPU bandwidths for all active tasks to exceed the available CPU bandwidth, then the new task will be rejected. This conservative approach overestimates the CPU bandwidth actually consumed by aperiodic tasks, which often use much less CPU time than their specifications would indicate. Thus, real-time admission control often rejects tasks that could actually be successfully supported, pessimistically reducing system utilization.

To increase system utilization, Redline uses a more permissive admission control policy. If the CPU bandwidth actually consumed by interactive tasks is not too high, then new interactive tasks may be admitted. Because this approach could lead to an overloaded system if interactive tasks begin to consume more CPU bandwidth, Redline employs load control that allows it to quickly recover from such overload. Consequently, Redline does not provide the inviolable guarantees of a real-time system, but in exchange for allowing short-lived system overloads, Redline ensures far higher utilization than real-time systems could provide.

To maximize utilization while controlling load, Redline strives to keep the CPU bandwidth consumed by interactive tasks within a fixed range. It tracks the actual CPU

load that drives its admission and load control policies. We describe here how Redline tracks load and how it manages interactive tasks.

Load tracking: Redline maintains two values that reflect CPU load. The first, R_{load}, represents the actual, recent CPU use. Once per second, Redline measures the CPU bandwidth consumed during that interval by interactive tasks. R_{load} is the arithmetic mean of the four most recent CPU bandwidth samples. This four-second *observation window* is long enough to smooth short-lived bursts of CPU use, and is short enough that longer-lived overloads are quickly detected.

The second value, S_{load}, projects expected CPU load. When an interactive task i is launched, the task's specified CPU bandwidth, $B_i = \frac{C_i}{T_i}$, is added to S_{load}. Since specifications are conservative, S_{load} may overestimate the load. Therefore, over time, S_{load} is exponentially decayed toward R_{load}. Additionally, as interactive tasks terminate, their contribution to S_{load} is subtracted from that value, thus updating the projection.

Management policies: Redline uses these CPU load values to control interactive tasks through a set of three policies:

Admission: When a new interactive task i is submitted, admission control must determine whether accepting the task will force the CPU load above a threshold R_{hi}, thus placing too high a load on the system. If $max(R_{load}, S_{load}) + B_i < R_{hi}$, then i is admitted as an interactive task; otherwise it is placed into the best-effort class. Thus, Redline avoids overloading the system, but does so based on measured system load, thus allowing higher utilization than real-time systems.

Revocation: Because of Redline's permissive admission control, it is possible for some interactive tasks to increase their CPU consumption and overload the system. Under these circumstances, Redline revokes the interactive classification of some tasks, demoting them to the best-effort class, thus allowing the remaining interactive tasks to remain responsive.

Specifically, if $R_{load} > R_{max}$, where $R_{max} > R_{hi}$, then Redline revokes tasks until R_{load} falls below R_{hi}. Redline prefers to revoke a task that exhausted its reservation during the observation window, indicating that the task may be more CPU-bound and less interactive. However, if there are no such tasks, Redline revokes the task with the highest CPU bandwidth consumption. Certain tasks are set to be invulnerable to revocation to preserve overall system responsiveness, such as kernel threads and the graphical user interface tasks.

Reactivation: If $R_{load} < R_{lo}$, Redline will reactivate previously revoked tasks, promoting them from the best-effort class to the interactive class. A task is eligible for reactivation if *both* (a) it passes the usual admission test, *and* (b) its virtual memory size minus its resident size is less than free memory currently available (i.e., it will not immediately induce excessive swapping). We further constrained Redline to reactivate only one task per period of an observation window to avoid reactivating tasks too aggressively.

6.2 The EDF Scheduling Algorithm

Redline extends the Linux fair queueing proportional share scheduler, *CFS* [20]. Redline's EDF scheduler has its own set of per-CPU run queues just as the CFS scheduler does. Redline inserts interactive tasks into the run queues of *both* the CFS and EDF schedulers so that each interactive task can receive any unused CPU bandwidth after consuming its reserved bandwidth. The EDF scheduler has precedence over the CFS scheduler. During a context switch, Redline first invokes the EDF scheduler. If the EDF scheduler has no runnable tasks to schedule, then Redline invokes the CFS scheduler.

Redline maintains the following information for each interactive task: the *starttime* and *deadline* of the current reservation period, and the remaining entitled computation time (*budget*). As a task executes, the EDF scheduler keeps track of its CPU usage and deducts the amount consumed from *budget*. The EDF scheduler checks whether to assign a new reservation period to a task at the following places: when a new task is initialized, after a task's budget is updated, and when a task wakes up from sleep. If the task has consumed its budget or passes its deadline, the EDF scheduler assigns a new reservation period to the task using the algorithm in Listing 1.

Listing 1 Assign a new reservation period to task p

```
1:  /* has budget, deadline not reached */
2:  if (budget > 0) && (now < deadline) then
3:      return
4:  end if
5:  /* has budget, deadline is reached */
6:  if (budget > 0) && (now ≥ deadline) then
7:      if has no interruptible sleep then
8:          return
9:      end if
10: end if
11:
12: /* no budget left: assign a new period */
13: dequeue(p)
14: starttime ← max(now,deadline)
15: deadline ← starttime + T
16: budget ← max(budget + C, C)
17: enqueue(p)
```

A task may reach its deadline before expending the budget (see line 6) for the following reasons: it did not actually have enough computation work to exhaust the budget in the past period; it ran into a CPU overload; or it experienced non-discretionary delays, such as page faults or disk I/O.

The EDF scheduler differentiates these cases by checking whether the task voluntarily gave up the CPU during the past period (i.e., had at least one interruptible sleep, see line 7). If so, the EDF scheduler assigns a new period to the task. Otherwise, it considers that the task missed a deadline and pushes its work through as soon as possible.

If a task consumes its budget before reaching the deadline, it will receive a new reservation period. But the start time of this new period is later than the current time (see line 14). The EDF scheduler considers a task *eligible* for using reserved CPU time only if $starttime \leq now$. Therefore, it will not pick this task for execution until its new reservation period starts. This mechanism prevents a interactive task from consuming more than its entitlement and thereby interfering with other interactive tasks.

At any context switch, the EDF scheduler always picks for execution the eligible task that has the earliest deadline. We implemented its run queue using a tagged red-black tree similar to the binary tree structure proposed in EEVDF [28]. The red-black tree is sorted by the start time of each task. Each node in the tree has a tag recording the earliest deadline in its subtree. The complexity of its enqueue, dequeue, and select operations are all $O(\log n)$, where n is the number of runnable tasks.

6.3 SMP Load Balancing

Load balancing in Redline is quite simple, because the basic CPU bandwidth needs of an interactive task will be satisfied once it is accepted on a CPU. It is not necessary to move an accepted task unless that CPU is overloaded. The only thing Redline has to do is select a suitable CPU for each new interactive task during calls to exec(). Redline always puts a new task on the CPU that has the lowest R_{load} at the time. Once the task passes the admission test, it stays on the same CPU as long as it remains accepted. If the CPU becomes overloaded, Redline will revoke at least one interactive task tied to that CPU. Once turned into best-effort tasks, revoked tasks can be moved to other CPUs by the load balancer. When a revoked task wakes up on a new CPU, Redline will attempt to reactivate its specification if there are adequate resources there.

7 Discussion

In this section, we discuss several design choices for Redline, and we address alternatives that may merit further investigation.

Specifications: Setting specifications in Redline does not require precise, *a priori* application information. Since Redline's admission control takes the current CPU load into account, a user or system administrator can modestly over-specify a task's resource needs. The only danger of over-specification is that a newly launched task may be rejected by the admission control if the system is sufficiently loaded with other interactive tasks. Once admitted, an interactive task is managed according to its real usage, negating the impact of the over-specification. If an interactive task is under-specified, it will at least make steady progress with the reserved CPU bandwidth allocated to it. Furthermore, if the system is not heavily loaded, an under-specified interactive task will also allocated some of the remaining CPU bandwidth along with the best-effort tasks. Thus, even under-specified tasks become poorly responsive only if the system load is high.

We therefore believe that specification management should not be an obstacle. A simple tool could allow a user to experimentally adjust the reservations (both C and T), finding minimal requirements for acceptable performance. Redline could also be made to track and report actual CPU bandwidth usage over time, allowing fine-tuning of the specifications.

Memory Management: For any task to be responsive, the operating system must cache its working set in main memory. Many real-time systems conservatively "pin" all pages of a task, preventing their removal from main memory to ensure that the working set must be cached. In contrast, Redline protects any page used by an interactive task within the last π seconds, thus protecting the task's working set while allowing its inactive pages to be evicted. In choosing a value for π, it is important not to make it too small, causing the system to be behave like a standard, commodity OS. It is safer to overestimate π, although doing so makes Redline behave more like a conservative real-time system. This method of identifying the working set works well under many circumstances with reasonable choices of π.

However, there are other mechanisms that estimate the working set more accurately. By using such mechanisms, Redline could avoid the dangers caused by setting π poorly. One such alternative is the working set measurement used in CRAMM [32]. CRAMM maintains reference distribution histograms to track each task's working set on-line with low overhead and high accuracy. While we believe that this approach is likely to work well, it does not guarantee that the working set is always properly identified. Specifically, if a task performs a *phase change*, altering its reference behavior suddenly and significantly, this mechanism will require a moderate period of time to recognize the change. During this period, CRAMM could substantially under- or over-estimate application working set sizes. However, we believe such behavior is likely to be brief and tolerable in the vast majority of cases. We intend to integrate the CRAMM VMM into Redline and evaluate its performance.

8 Experimental Evaluation

Figure 1 shows that the execution of a single compilation command (make -j32) significantly degrades responsiveness on a standard system, while Redline is able to keep the system responsive and play the video smoothly. In this

section, we further evaluate the performance of Redline implementation by stressing the system with a variety of extreme workloads.

Platform: We perform all measurements on a system with a 3 GHz Pentium 4 CPU, 1 GB of RAM, a 40GB FUJITSU 5400RPM ATA disk, and an Intel 82865G integrated graphics card. The processor employs *symmetric multithreading (SMT)* (i.e., Intel's HyperThreading), thus appearing to the system as two processors. We use a Linux kernel (version 2.6.22.5) patched with the CFS scheduler (version 20.3) as our control. Redline is implemented as a patch to this same Linux version. For all experiments, the screen resolution was set to 1600 x 1200 pixels.

All experiments used both of the SMT-based virtual CPUs except when measuring the context switch overhead. Furthermore, we ran each experiment 30 times, taking both the arithmetic mean and the standard deviation of all timing measurements.

Application Settings and Inputs: Table 1 shows a subset of the task specifications used for the experiments under Redline. It includes the *init* process, *kjournald*, the X11 server *Xorg*, KDE's desktop/window manager, the *bash* shell, and several typical interactive applications. We left the memory protection period (π) and I/O priority empty in all the specifications, letting Redline choose them automatically.

	C:T (ms)		C:T (ms)
init	2:50	kjournald	10:100
Xorg	15:30	kdeinit	2:30
kwin	3:30	kdesktop	3:30
bash	5:100	vim	5:100
mplayer	5:30	firefox	6:30

Table 1: A subset of the specifications used in the Redline experiments.

The movie player, *mplayer*, plays a 924.3 Kb/s AVI-format video at 25 frames per second (f/s) with a resolution of 520 x 274. To give the standard Linux system the greatest opportunity to support these interactive tasks, we set *mplayer*, *firefox*, and *vim* to have a CPU scheduler priority of -20—the highest priority possible. Also note that a pessimistic admission test, like that of a real-time system, would not accept all of the applications in Table 1, because they would over-commit the system. Redline, however, accepts these as well as many other interactive tasks.

8.1 CPU Scheduling

Scheduler Overhead: We compare the performance of the Redline EDF scheduler with the Linux CFS scheduler. Figure 4(a) presents the context switch overhead for each scheduler as reported by *lmbench*. From 2 to 96 processes, the context switch time for both schedulers is exceedingly comparable.

However, when lmbench measures context switching time, it creates a workload in which exactly one task is unblocked and ready to run at any given moment. This characteristic of lmbench may be unrepresentative of some workloads, so we further compare these schedulers by running multiple busy-looping tasks, all of which would be ready to run at any moment. We tested workloads of 1, 20, 200, and 2,000 tasks. We perform this test twice for Redline, launching best-effort tasks to test its CFS scheduler, and launching interactive tasks to test its EDF scheduler. In the latter case, we assigned specification values for C:T as 1:3, 1:30, 1:300 and 1:3,000 respectively, thus forcing the Redline EDF scheduler to perform a context switch almost every millisecond. Note that with these specification values, the Redline EDF scheduler is invoked more frequently than a CFS scheduler would be, running roughly once per millisecond (whereas the usual Linux CFS quanta is 3 ms).

The number of loops are chosen so that each run takes approximately 1,400 seconds. Figure 4(b) shows the mean total execution time of running these groups of tasks with Linux CFS, Redline CFS, and Redline EDF. In the worst case, the Redline EDF scheduler adds 0.54% to the running time, even when context switching far more frequently than the CFS schedulers. Note that the total execution time of these experiments was bimodal and, as shown by the error bars, made the variance in running times larger than the difference between results. The overhead of using the Redline schedulers is negligible.

Fork Bombs: We now evaluate the ability of Redline to maintain the responsiveness of interactive tasks. First, we launch mplayer as an interactive task, letting it run for a few seconds. Then, we simultaneously launch many CPU-intensive tasks, thus performing a fork bomb. Specifically, a task forks a fixed number of child tasks, each of which executes an infinite loop, and then kills them after 30 seconds. We performed two tests for Redline: the first runs the fork bomb tasks as best-effort, and the second runs them as interactive. In the latter case, the fork bomb tasks were given CPU bandwidth specifications of 10:100. For the Linux test, the interactive task had the highest possible priority (-20), while the fork bomb tasks were assigned the default priority (0).

Figure 5 shows the number of frames rate of achieved by mplayer during these tests. In Figure 5(a), the fork bomb comprises 50 tasks, while Figure 5(b) shows a 2,000-task fork bomb. Both of these figures show, for Redline, best-effort and interactive fork bombs. Under Linux with 50 tasks, mplayer begins normally. After approximately 10 seconds, the fork bomb begins and mplayer receives so little CPU bandwidth that its frame rate drops nearly to zero. Amusingly, after the fork bomb terminates at the 40 second mark, mplayer "catches up" by playing frames at more than triple the normal rate. For Linux, the 2,000-task fork bomb has the identical effect on mplayer. The load is so high that

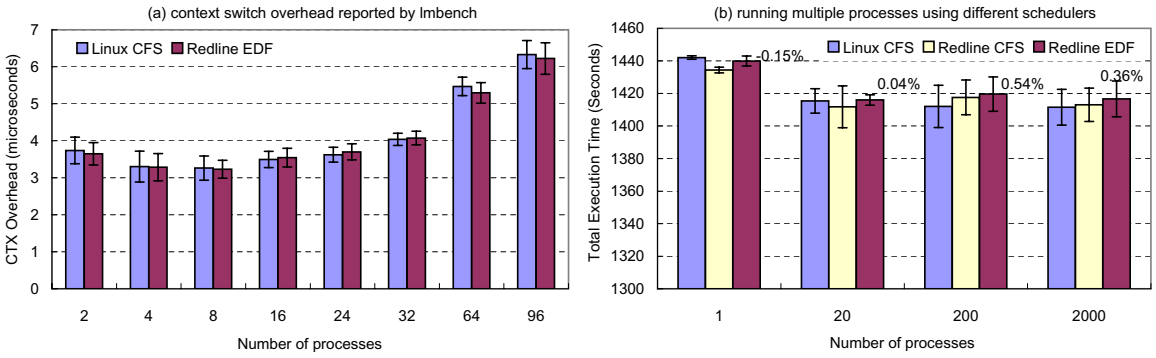


Figure 4: An evaluation of CPU scheduling overhead. Figure (a) shows the context switching times as evaluated by *lmbench*. Figure (b) shows the total running time of varying numbers of CPU intensive tasks. Note the y-axis starting at 1,300 to make the minor variation visible.

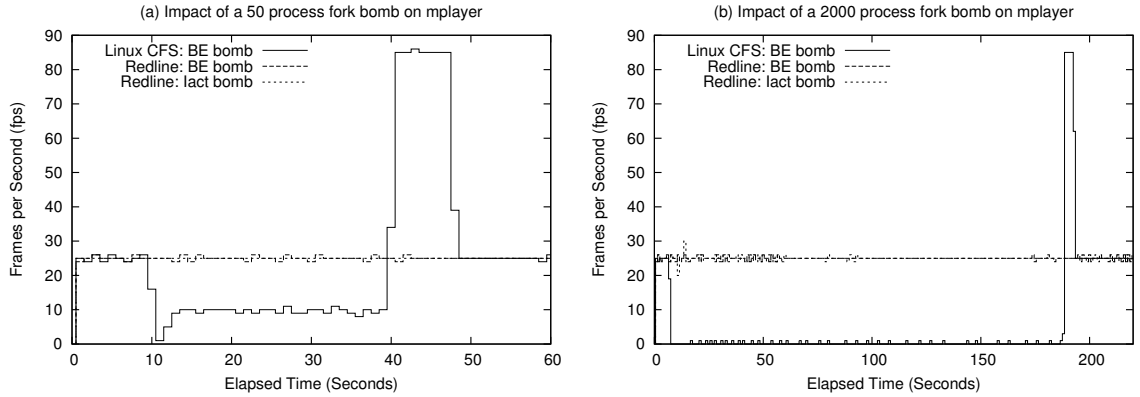


Figure 5: Playing a video and launching fork bombs of (a) 50 or (b) 2,000 (b) tasks.

even the fork bomb's parent task is unable to kill all of the children after 30 seconds, delaying the "catch-up" phase of mplayer until approximately the 190 second mark. In fact, the whole Linux system becomes unresponsive, with simple tasks like moving the mouse and switching between windows becoming so slow that human intervention is impossible.

In Redline, these fork bombs, whether run as best-effort or interactive tasks, have a negligible impact on mplayer. Only the 2,000 task interactive fork bomb briefly degrades the frame rate to 20 f/s, which is a barely perceptible effect. This brief degradation is caused by the 1 ms period that Redline gives to each newly forked task before performing an admission test, leaving the system temporarily overloaded.

Competing Interactive Tasks: In order to see how interactive tasks may affect each other, we launch mplayer, and then we have a user drag a window in a circle for 20 seconds. Figure 6 shows that, under Linux, moving a window has substantial impact on mplayer. Because we use a high screen resolution and a weakly powered graphics card,

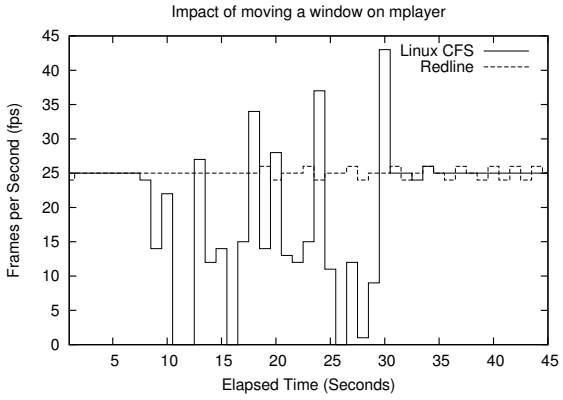


Figure 6: Playing video while dragging around a window.

Xorg requires a good deal of CPU bandwidth to update the screen. However, the CFS scheduler gives the same CPU share to all runnable tasks, allowing the window manager to submit screen update requests faster than Xorg can process them. When mplayer is awakened, it has to wait until

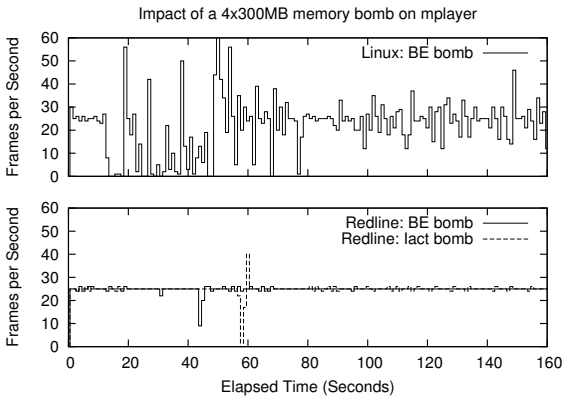


Figure 7: Playing video with 4 x 300 MB memory bomb tasks. The frame rate is severely erratic under Linux, but is steady under Redline.

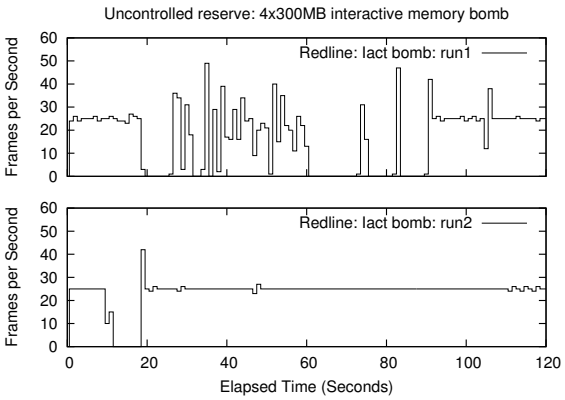


Figure 8: Playing a movie with 4 x 300 MB interactive memory bomb tasks on a Redline system without the rate controlled memory reserve.

all other runnable tasks make sufficient progress before it is scheduled for execution. Moreover, its requests are inserted at the end of Xorg's backlogged service queue. Consequently, the frame rate of mplayer becomes quite erratic as it falls behind and then tries to catch up by submitting a group of frame updates in rapid succession.

In Redline, mplayer plays the movie smoothly no matter how quickly we move the window, even though Xorg and all of the tasks comprising the GUI are themselves interactive tasks. We believe that because Xorg effectively gets more bandwidth (50% reserved plus proportional sharing with other tasks), and the EDF scheduler causes mplayer add its requests into Xorg's service queue earlier.

8.2 Memory Management

Memory Bombs: We simulate a workload that has a high memory demand by using memory bombs. This experiment creates four tasks, each of which allocates 300 MB of heap space and then repeatedly writes to each page in an infinite loop. For Redline, we perform two experiments: the first launches the memory bombs as best-effort tasks, and the second launches them as interactive ones using specification 10:100.

The upper part of Figure 7 shows, for Linux, the frame rate for mplayer over time with the memory bomb tasks running. The frame rate is so erratic that movie is unwatchable. Both video and audio pause periodically. The memory bomb forces the VMM to swap out many pages used by GUI applications, making the system as a whole unresponsive. As shown by the lower part of Figure 7, under Redline, mplayer successfully survives both the best-effort and interactive memory bombs. Each of them only leads to one brief disruption of the frame rate (less than 3 seconds), which a user will notice but is likely to tolerate. The system remains responsive, allowing the user to carry out GUI operations and interact as usual.

The Rate Controlled Reserve: In order to demonstrate the importance of the rate controlled reserve, we remove it from Redline and repeat the interactive memory bomb experiment. Figure 8 shows how mplayer behaves in two different runs. In the first, memory-demanding interactive tasks quickly exhaust the free memory, forcing others tasks to reclaim pages when allocating memory. Therefore, mplayer is unable to maintain its frame rate. At approximately the 90 second mark, the Redline VMM finally demotes an interactive task to the best-effort class, and then the frame rate of mplayer stabilizes. Depending on when and which tasks the Redline VMM chooses to revoke, the interactive memory bomb can prevent the system from being responsive for a long period of time. Here, more than one minute passes before responsiveness is restored. However, during the second run, mplayer was unresponsive for only about 10 seconds, thanks to a different selection of tasks to demote. Ultimately, the limited isolation among interactive tasks provided by this small rate controlled reserve is crucial to the maintaining a consistently responsive system.

Speed-bump Pages: To examine the effectiveness of Redline's speed-bump page mechanism, we first start one 500 MB memory bomb task. After a few seconds, we launch a second 500 MB memory bomb. Under Linux, we set this second task's priority to be -20. Under Redline, we launch it as an interactive task whose specification is set to 10:100. Figure 9 presents the *resident set sizes (RSS)*—the actual number of cached pages—for each task over time. Under Linux, the second task, in spite of its high priority, is never allocated its complete working set of 500 MB. Here, the first task dirties pages too fast, preventing the Linux VMM from ever reallocating page frames to the higher priority task. However, under Redline, the second task is quickly allocated space for its full working set, stealing pages from the first, best-effort task.

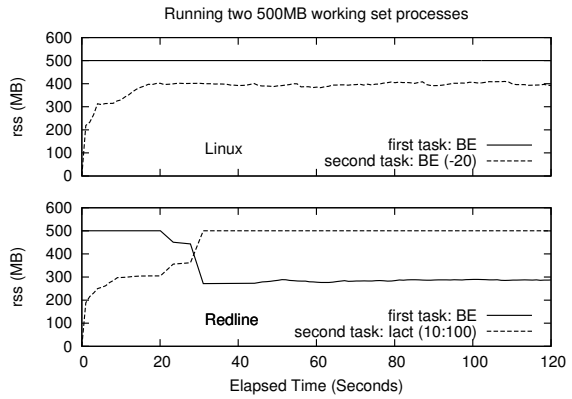


Figure 9: Competing memory bomb tasks. Under Linux, the lower-priority background task prevents the higher-priority foreground task from caching its working set.

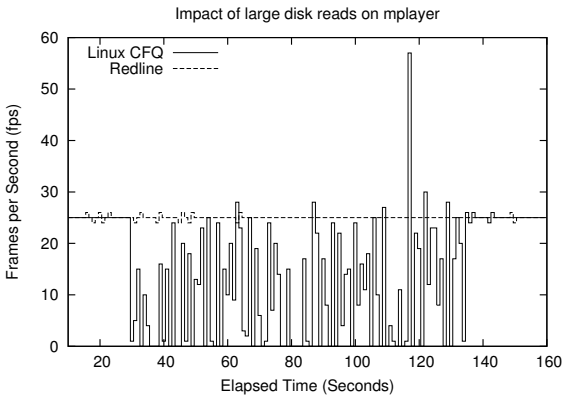


Figure 10: The impact of massive reads on mplayer.

As an alternative test of this mechanism, we launched four 300 MB memory bombs, and then launched the firefox web browser. Under Redline, firefox was ready and responsive after 30 seconds; under a standard Linux kernel, more than 4 minutes passed before the browser could be used.

8.3 Disk I/O Management

We examine the effectiveness of Redline's disk I/O management by running tasks that perform intensive disk I/O.

Writing: To test disk writing operations, we launch two background tasks meant to interfere with responsiveness. Specifically, each task repeatedly writes to an existing, 200 MB file using buffered writes. We then use vim, modified to report the time required to execute its write command, to perform sporadic write requests of a 30 KB file. It is set to the highest priority (-20) under Linux, while it is launched as an interactive task with a specification of 5:100 under Redline.

For Linux, each transaction in the journaling file system is heavily loaded with dirty pages from the background tasks. Thus, the calls to fsync() performed by vim causes it to block for a mean of 28 seconds. Under Redline, the reduced dirty threshold for best-effort tasks forces the system to flush the dirtied pages of the background task more frequently. When vim calls fsync(), the transaction committed by the journaling file system takes much less time because it is much smaller, requiring a mean of only 2.5 seconds.

Reading: We play a movie using mplayer in the foreground while nine background tasks consume all of the disk bandwidth. Each background task reads 100 MB from disk in 20 MB chunks using direct I/O (bypassing the file system cache to achieve steady I/O streams). Figure 10 shows the number frame rate of mplayer over time for both Linux and Redline. Under Linux, mplayer blocks frequently because of the pending I/O requests of the background tasks, thus making the frame rate severely degraded and erratic. Redline automatically assigns higher I/O priorities to the interactive task, allowing it to maintain its framerate throughout.

9 Related Work

We now briefly discuss related work in the areas of scheduling, virtual memory management, and I/O management. Table 2 summarizes the characteristics of several representative CPU schedulers and operating systems, and compares them to Redline.

CPU Scheduling: Neither time-sharing schedulers (e.g., used by Linux, FreeBSD, Solaris, Windows) nor proportional-share schedulers [10, 31, 27, 28, 22] that simulate the ideal GPS model [24] provide adequate support to address response time requirements. These systems typically employ heuristics to improve the responsiveness of interactive tasks. For example, FreeBSD boosts the priorities for I/O-bound tasks, and Windows Vista does the same for tasks running in its multimedia class.

Several extended general-purpose schedulers provide performance isolation across scheduling classes. A-SFQ [26] dynamically adjusts weights to maintain stable CPU bandwidth for the class serving soft real-time tasks. BVT [8] and BERT [3] achieve the same goal by adjusting virtual time or deadlines, while BEST [1] and SMART [21] incorporate EDF into proportional share schedulers. However, without appropriate admission and load control, none of these systems can provide CPU bandwidth guarantees under heavy workloads.

Unlike these general-purpose systems, real-time systems often impose draconian admissions control and enforcement mechanisms in order to provide strict performance guarantees. Examples include Nemesis [16], CPU service class [6], Linux/RK [23], Rialto [15], Lin et al.'s scheduler [11], and work by Deng et al. [7] and PSheED [17]. While specification-based admission tests pessimistically reject tasks in exchange for strict guarantees, Redline adapts to the actual system workload, maximizing resource

		Admission control	Performance isolation		Intg. Mgmt.		Without app. mod.
			interclass	intraclass	mem	I/O	
CPU Scheduler	Stride [31],EEVDF [28], VTRR [22],PD [27]	×	×	×	n/a	n/a	√
	SFQ [10], A-SFQ [26]	×	strong	×	n/a	n/a	√
	BVT [8], BERT [3]	×	strong	weak	n/a	n/a	×
	BEST [1]	×	weak	weak	n/a	n/a	√
	SMART [21]	×	strong	strong	n/a	n/a	×
	PSheED [17], Deng et al. [7]	pessimistic	strict	strict	n/a	n/a	×
Operating Systems	Linux, FreeBSD, Solaris, Windows	×	×	×	×	×	√
	Solaris Container [19], Eclipse [4], SPU [30]	×	strong	×	√	√	√
	QLinux [29]	×	strong	×	×	√	√
	Linux-SRT [5]	pessimistic	strong	×	×	√	×
	Rialto [15]	pessimistic	strict	strict	×	√	×
	Nemesis [16]	pessimistic	strict	strict	×	√	×
	Redline	load based	strong	dynamic	√	√	√

Table 2: A comparison of Redline to other CPU schedulers and operating systems

utilization while accommodating as many interactive tasks as possible.

Memory Management: A number of memory managers employ strategies that improve on the traditional global LRU model still used by Linux. The Windows virtual memory manager adopts a per-process working set model, with a kernel thread that attempts to move pages from each process's working set onto a *standby* reclamation list, prioritized based on task priorities. In Zhou et al. [33], the virtual memory manager maintains a miss ratio curve for each process and evicts pages from the process that incurs the least penalty. Token-ordered LRU [14] allows *one* task in the system to hold a token for a period of time to build up its working set. CRAMM [32] and Bookmarking GC [12] use operating system support that allows garbage collected applications avoid page swapping. Unlike Redline, none of these memory managers can prevent the system from evicting too many pages from interactive applications.

Disk I/O Management: Traditional disk I/O subsystems are designed to maximize the overall throughput, rather than minimizing response time. The widely used SCAN (Elevator) algorithm sorts I/O requests by sector number to avoid unnecessary seeks. Anticipatory I/O [13] improves throughput even further by delaying I/O service so it can batch a number of I/O requests. Some operating systems (e.g. Linux and Windows) can prioritize I/O requests according to task priorities. This mechanism has little effect in practice, since most applications are launched with the same default priority. While Redline's I/O scheduler is effective, I/O schedulers developed for soft real time systems, such as R-SCAN in Nemesis [16], Cello in QLinux [29] and DS-SCAN in IRS [9], could be incorporated into Redline to enable more precise control over I/O bandwidth.

Integrated Resource Management: Like Redline, Nemesis [16], Rialto [15] and Linux-SRT [5] use *CPU reservations* to provide response time guarantees and include specialized I/O schedulers. However, both systems employ pessimistic admission policies that reduce their ability to handle a large number of aperiodic tasks with unpredictable workloads. More importantly, these systems do not address the thorny issue of memory management: Nemesis allocates a certain amount of physical memory to each task according to its contract and lets each task manage its own memory via *self-paging*, while Rialto simply locks all memory pages used by real time tasks. Similarly, QLinux [29] divides tasks into classes and uses a hierarchical SFQ scheduler to manage CPU and network bandwidth together with Cello as its I/O scheduler, but does not address the problem of memory management.

Solaris Containers [19] combine system resource management with a boundary separation provided by *zones*. Each zone can be configured to have a dedicated amount of resources and then acts as a completely isolated virtual server. Eclipse [4] and SPU [30] follow a similar approach, partitioning resources *statically* and ignoring the issue of responsiveness. Moreover, it is highly impractical to create one zone/container for every possible interactive application due to complex interactions among them and the tremendous configuration effort that would be required. By contrast, Redline's adaptive approach is simpler and better matches the needs of supporting interactive applications in commodity operating systems.

IRS [9] and Resource Containers [2] also provide integrated management of multiple resources with a focus on providing real time service to multiple clients, although IRS does not address memory management. However, unlike Redline, these systems require that all applications in a system explicitly state their resource needs, requiring complex modifications to existing applications.

10 Conclusion

We present Redline, a system designed to support highly interactive applications in a commodity operating system environment. Redline combines lightweight specifications with an integrated management of memory, disk I/O, and CPU resources that delivers responsiveness to interactive applications even in the face of extreme workloads.

The Redline system (built on Linux) is open-source software and may be downloaded at `http://redline.cs.umass.edu`.

11 Acknowledgements

This material is based upon work supported by the National Science Foundation under CAREER Award CNS-0347339 and CNS-0615211. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] S. A. Banachowski and S. A. Brandt. Better real-time response for time-share scheduling. In *Proc. of the* 11^{th} *WPDRTS*, page 124.2, 2003.

[2] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. of the* 3^{rd} *OSDI*, pages 45–58, 1999.

[3] A. Bavier, L. Peterson, and D. Mosberger. BERT: A scheduler for best effort and realtime tasks. Technical Report TR-587-98, Princeton University, 1999.

[4] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz. The Eclipse operating system: Providing quality of service via reservation domains. In *Proc. of the 1998 USENIX*, pages 235–246, 1998.

[5] S. Childs and D. Ingram. The Linux-SRT integrated multimedia operating system: Bringing QoS to the desktop. In *Proc. of the* 7^{th} *RTAS*, pages 135–140, 2001.

[6] H.-H. Chu and K. Nahrstedt. CPU service classes for multimedia applications. In *Proc. of the* 6^{th} *ICMCS, Vol. 1*, pages 296–301, 1999.

[7] Z. Deng, J. Liu, L. Y. Zhang, M. Seri, and A. Frei. An open environment for real-time applications. *Real-Time Systems*, 16(2-3):155–185, 1999.

[8] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose schedular. In *Proc. of the* 17^{th} *SOSP*, pages 261–276, 1999.

[9] K. Gopalan and T. Chiueh. Multi-resource allocation and scheduing for periodic soft real-time applications. In *Proc. of the* 9^{th} *MMCN*, pages 34–45, Berkeley, CA, 2002.

[10] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proc. of the* 2^{nd} *OSDI*, pages 107–121, Seattle, WA, 1996.

[11] C. han Lin, H. hua Chu, and K. Nahrstedt. A soft real-time scheduling server on the Windows NT. In *Proc. of the* 2^{nd} *USENIX Windows NT Symposium*, pages 149–156, 1998.

[12] M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. In *Proc. of the 2005 PLDI*, pages 143–153, 2005.

[13] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proc. of the* 18^{th} *SOSP*, pages 117–130, 2001.

[14] S. Jiang and X. Zhang. Token-ordered LRU: an effective page replacement policy and its implementation in Linux systems. *Perform. Eval.*, 60(1-4):5–29, 2005.

[15] M. B. Jones, D. L. McCulley, A. Forin, P. J. Leach, D. Rosu, and D. L. Roberts. An overview of the Rialto real-time architecture. In *Proc. of the* 7^{th} *ACM SIGOPS European Workshop*, pages 249–256, 1996.

[16] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, 1996.

[17] G. Lipari, J. Carpenter, and S. K. Baruah. A framework for achieving inter-application isolation in multiprogrammed hard real-time environments. In *Proc. of the* 21^{st} *RTSS*, pages 217–226, 2000.

[18] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of ACM*, 20(1):46–61, 1973.

[19] J. Mauro and R. McDougall. *Solaris Internal: Core Kernel Components*. Sum Microsystems Press, A Prentice Hall Title, 2000.

[20] I. Molnar. http://people.redhat.com/mingo/cfs-scheduler/.

[21] J. Nieh and M. S. Lam. SMART: A processor scheduler for multimedia applications. In *Proc. of the* 15^{th} *SOSP*, page 233, 1995.

[22] J. Nieh, C. Vaill, and H. Zhong. Virtual-Time Round-Robin: An O(1) proportional share scheduler. In *Proc. of the 2001 USENIX*, pages 245–259, 2001.

[23] S. Oikawa and R. Rajkumar. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Proc. of the* 5^{th} *RTAS*, pages 111–120, 1999.

[24] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single node case. In *Proc. of IEEE INFOCOM*, 1992.

[25] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proc. of the 2005USENIX*, pages 105–120, 2005.

[26] M. A. Rau and E. Smirni. Adaptive CPU scheduling policies for mixed multimedia and best-effort workloads. In *Proc. of the* 7^{th} *MASCOTS*, page 252, Washington, DC, 1999.

[27] A. Srinivasan and J. H. Anderson. Fair scheduling of dynamic task systems on multiprocessors. *Journal of System Software*, 77(1):67–80, 2005.

[28] I. Stoica and H. Abdel-Wahab. Earliest eligible virtual deadline first : A flexible and accurate mechanism for proportional share resource allocation. Technical Report TR-95-22, Old Dominion University, 1995.

[29] V. Sundaram, A. Chandra, P. Goyal, P. J. Shenoy, J. Sahni, and H. M. Vin. Application performance in the QLinux multimedia operating system. In *Proc. of the* 8^{th} *ACM Multimedia*, pages 127–136, 2000.

[30] B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *Proc. of the* 8^{th} *ASPLOS*, pages 181–192, 1998.

[31] C. A. Waldspurger and W. E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical Report TR-528, MIT Laboratory of CS, 1995.

[32] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *Proc. of the* 7^{th} *OSDI*, pages 103–116, 2006.

[33] P. Zhou, V. Pandy, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curves for memory management. In *Proc. of the* 11^{th} *ASPLOS*, pages 177–188, Boston, MA, Oct. 2004.

Network Imprecision: A New Consistency Metric for Scalable Monitoring

Navendu Jain[†], Prince Mahajan[⋆], Dmitry Kit[⋆], Praveen Yalagandula[‡], Mike Dahlin[⋆], and Yin Zhang[⋆]
[†]*Microsoft Research* [⋆]*The University of Texas at Austin* [‡]*HP Labs*

Abstract

This paper introduces a new consistency metric, *Network Imprecision* (NI), to address a central challenge in large-scale monitoring systems: safeguarding accuracy despite node and network failures. To implement NI, an overlay that monitors a set of attributes also monitors its own state so that queries return not only attribute values but also information about the stability of the overlay—the number of nodes whose recent updates may be missing and the number of nodes whose inputs may be double counted due to overlay reconfigurations. When NI indicates that the network is stable, query results are guaranteed to reflect the true state of the system. But when the network is unstable, NI puts applications on notice that query results should not be trusted, allowing them to take corrective action such as filtering out inconsistent results. To scalably implement NI's introspection, our prototype introduces a key optimization, dual-tree prefix aggregation, which exploits overlay symmetry to reduce overheads by more than an order of magnitude. Evaluation of three monitoring applications demonstrates that NI flags inaccurate results while incurring low overheads, and monitoring applications that use NI to select good information can improve their accuracy by up to an order of magnitude.

1 Introduction

Scalable system monitoring is a fundamental abstraction for large-scale networked systems. It enables operators and end-users to characterize system behavior, from identifying normal conditions to detecting unexpected or undesirable events—attacks, configuration mistakes, security vulnerabilities, CPU overload, or memory leaks—before serious harm is done. Therefore, it is a critical part of infrastructures ranging from network monitoring [10,23,30,32,54], financial applications [3], resource scheduling [27, 53], efficient multicast [51], sensor networks [25, 27, 53], storage systems [50], and bandwidth provisioning [15], that potentially track thousands or millions of dynamic attributes (e.g., per-flow or per-object state) spanning thousands of nodes.

Three techniques are important for scalability in monitoring systems: (1) *hierarchical aggregation* [27, 30, 51, 53] allows a node to access detailed views of nearby information and summary views of global information, (2) *arithmetic filtering* [30, 31, 36, 42, 56] caches recent reports and only transmits new information if it differs by some numeric threshold (e.g., $\pm$ 10%) from the cached report, and (3) *temporal batching* [32, 36, 42, 51] combines multiple updates that arrive near one another in time into a single message. Each of these techniques can reduce monitoring overheads by an order of magnitude or more [30, 31, 42, 53].

As important as these techniques are for scalability, they interact badly with node and network failures: a monitoring system that uses any of these techniques risks reporting highly inaccurate results.

- In a hierarchical monitoring system, the impact of failures is made worse by the *amplification effect* [41]: if a non-leaf node fails, then the entire subtree rooted at that node can be affected. For example, failure of a level-3 node in a degree-8 aggregation tree can interrupt updates from 512 (8^3) leaf node sensors.
- When a monitoring system uses arithmetic filtering, if a subtree or node is silent over an interval, the system must distinguish two cases: (a) the subtree or node has sent no updates because the inputs have not significantly changed from the cached values or (b) the inputs have significantly changed but the subtree or node is unable to transmit its report.
- Under temporal batching there are windows of time in which a short disruption can block a large batch of updates.

These effects can be significant. For example, in an 18-hour interval for a PlanetLab monitoring application, we observed that more than half of all reports differed from the ground truth at the inputs by more than 30%. These best effort results are clearly unacceptable for many applications.

To address these challenges, we introduce *Network Imprecision* (NI), a new consistency metric suitable for large-scale monitoring systems with unreliable nodes or networks. Intuitively, NI represents a "stability flag" indicating whether the underlying network is stable or not. More specifically, with each query result, NI provides (1) the number of nodes whose recent updates may not be reflected in the current answer, (2) the number of nodes whose inputs may be double counted due to overlay reconfiguration, and (3) the total number of nodes

in the system. A query result with no unreachable or double counted nodes is *guaranteed* to reflect reality, but an answer with many of either indicates a low system confidence in that query result—the network is unstable, hence the result should not be trusted.

We argue that NI's introspection on overlay state is the right abstraction for a *monitoring system* to provide to *monitoring applications*. On one hand, traditional data consistency algorithms [56] must block reads or updates during partitions to enforce limits on inconsistency [19]. However, in distributed monitoring, (1) updates reflect external events that are not under the system's control so cannot be blocked and (2) reads depend on inputs at many nodes, so blocking reads when any sensor is unreachable would inflict unacceptable unavailability. On the other hand, "reasoning under uncertainty" techniques [48] that try to automatically quantify the impact of disruptions on individual attributes are expensive because they require per-attribute computations. Further, these techniques require domain knowledge thereby limiting flexibility for multi-attribute monitoring systems [42, 51, 53], or use statistical models which are likely to be ineffective for detecting unusual events like network anomalies [26]. Even for applications where such application-specific or statistical techniques are appropriate, NI provides a useful signal telling applications when these techniques should be invoked.

NI allows us to construct PRISM (PRecision Integrated Scalable Monitoring), a new monitoring system that maximizes scalability via arithmetic filtering, temporal batching, and hierarchy. A key challenge in PRISM is implementing NI efficiently. First, because a given failure has different effects on different aggregation trees embedded in PRISM's scalable DHT, the NI reported with an attribute must be specific to that attribute's tree. Second, detecting missing updates due to failures, delays, and reconfigurations requires frequent active probing of paths within a tree. To provide a topology-aware implementation of NI that scales to tens of thousands of nodes and millions of attributes, PRISM introduces a novel *dual-tree prefix aggregation* construct that exploits symmetry in its DHT-based aggregation topology to reduce the per-node overhead of tracking the n distinct NI values relevant to n aggregation trees in an n-node DHT from $O(n)$ to $O(\log n)$ messages per time unit. For a 10K-node system, dual tree prefix aggregation reduces the per node cost of tracking NI from a prohibitive 1000 messages per second to about 7 messages per second.

Our NI design separates mechanism from policy and allows applications to use any desired technique to quantify and minimize the impact of disruptions on system reports. For example, in PRISM, monitoring applications use NI to safeguard accuracy by (1) inferring an approximate confidence interval for the number of sensor inputs contributing to a query result, (2) differentiating between correct and erroneous results based on their NI, or (3) correcting distorted results by applying redundancy techniques and then using NI to automatically select the best results. By using NI metrics to filter out inconsistent results and automatically select the best of four redundant aggregation results, we observe a reduction in the worst-case inaccuracy by up to an order of magnitude.

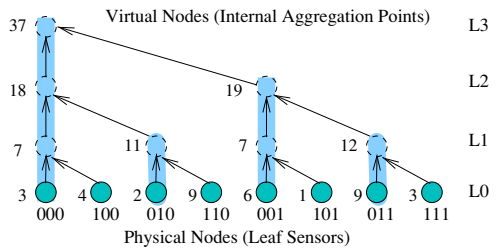


Figure 1: The aggregation tree for key 000 in an eight node system. Also shown are the aggregate values for a simple SUM() aggregation function.

This paper makes five contributions. First, we present Network Imprecision, a new consistency metric that characterizes the impact of network instability on query results. Second, we demonstrate how different applications can leverage NI to detect distorted results and take corrective action. Third, we provide a scalable implementation of NI for DHT overlays via dual-tree prefix aggregation. Fourth, our evaluation demonstrates that NI is vital for enabling scalable aggregation: a system that ignores NI can often silently report arbitrarily incorrect results. Finally, we demonstrate how a distributed monitoring system can both maximize scalability by combining hierarchy, arithmetic filtering, and temporal batching and also safeguard accuracy by incorporating NI.

2 Scalability vs. correctness

As discussed in Section 1, large-scale monitoring systems face two key challenges to safeguarding result accuracy. First, node failures, network disruptions, and topology reconfigurations imperil accuracy of monitoring results. Second, common scalability techniques—hierarchical aggregation [5, 44, 47, 53], arithmetic filtering [30, 31, 36, 38, 42, 51, 56], and temporal batching [14, 32, 36, 56]—make the problem worse. In particular, although each technique significantly enhances scalability, each also increases the risk that disruptions will cause the system to report incorrect results.

For concreteness, we describe PRISM's implementation of these techniques although the challenges in safeguarding accuracy are applicable to any monitoring system that operates under node and network failures. We compute a SUM aggregate for all the examples in this section.

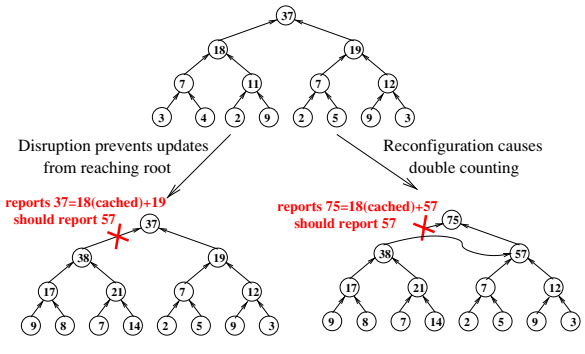


Figure 2: Dynamically-constructed aggregation hierarchies raise two challenges for guaranteeing the accuracy of reported results: the *failure amplification effect* and *double counting* caused by reconfiguration.

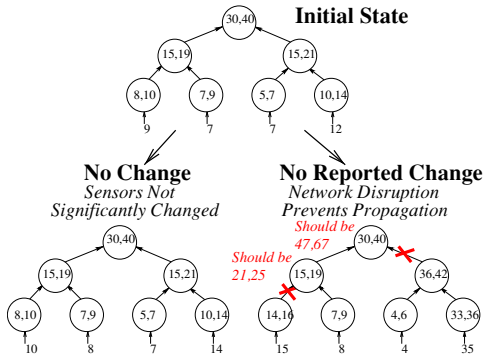


Figure 3: Arithmetic filtering makes it difficult to determine if a subtree's silence is because the subtree has nothing to report or is unreachable.

Hierarchical aggregation. Many monitoring systems use hierarchical aggregation [47, 51] or DHT-based hierarchical aggregation [5, 39, 44, 53] that defines a tree spanning all nodes in the system. As Figure 1 illustrates, in PRISM, each physical node is a leaf and each subtree represents a logical group of nodes; logical groups can correspond to administrative domains (e.g., department or university) or groups of nodes within a domain (e.g., a /28 IPv4 subnet with 14 hosts in the CS department) [22, 53]. An internal non-leaf node, which we call a *virtual node*, is emulated by a physical leaf node of the subtree rooted at the virtual node.

PRISM leverages DHTs [44, 46, 49] to construct a forest of aggregation trees and maps different attributes to different trees for scalability. DHT systems assign a long (e.g., 160 bits), random ID to each node and define a routing algorithm to send a request for ID i to a node $root_i$ such that the union of paths from all nodes forms a tree $DHTtree_i$ rooted at the node $root_i$. By aggregating an attribute with ID i = hash(attribute) along the aggregation tree corresponding to $DHTtree_i$, different attributes are load balanced across different trees. This approach can provide aggregation that scales to large numbers of nodes and attributes [5, 44, 47, 53].

Unfortunately, as Figure 2 illustrates, hierarchical aggregation imperils correctness in two ways. First, a failure of a single node or network path can prevent updates from a large collection of leaves from reaching the root, amplifying the effect of the failure [41]. Second, node and network failures can trigger DHT reconfigurations that move a subtree from one attachment point to another, causing the subtree's inputs to be double counted by the aggregation function for some period of time.

Arithmetic Imprecision (AI). Arithmetic imprecision deterministically bounds the difference between the reported aggregate value and the true value. In PRISM, each aggregation function reports a bounded numerical range $\{V_{min}, V_{max}\}$ that contains the true aggregate value V i.e., $V_{min} \leq V \leq V_{max}$.

Allowing such arithmetic imprecision enables arithmetic filtering: a subtree need not transmit an update unless the update drives the aggregate value outside the range it last reported to its parent; a parent caches last reported ranges of its children as soft state. Numerous systems have found that allowing small amounts of arithmetic imprecision can greatly reduce overheads [30, 31, 36, 42, 51, 56].

Unfortunately, as Figure 3 illustrates, arithmetic filtering raises a challenge for correctness: if a subtree is silent, it is difficult for the system to distinguish between two cases. Either the subtree has sent no updates because the inputs have not significantly changed from the cached values or the inputs have significantly changed but the subtree is unable to transmit its report.

Temporal Imprecision (TI). Temporal imprecision bounds the delay from when an event occurs until it is reported. In PRISM, each attribute has a TI guarantee, and to meet this bound the system must ensure that updates propagate from the leaves to the root in the allotted time.

As Figure 4 illustrates, TI allows PRISM to use temporal batching: a set of updates at a leaf sensor are condensed into a periodic report or a set of updates that arrive at an internal node over a time interval are combined before being sent further up the tree. Note that arithmetic filtering and temporal batching are complementary: a batched update need only be sent if the combined update drives a subtree's attribute value out of the range previously reported up the tree.

Of course, an attribute's TI guarantee can only be ensured if there is a *good path* from each leaf to the root. A good path is a path whose processing and propagation times fall within some pre-specified delay budget. Unfortunately, failures, overload, or network congestion can cause a path to no longer be good and prevent the system from meeting its TI guarantees. Furthermore, when a

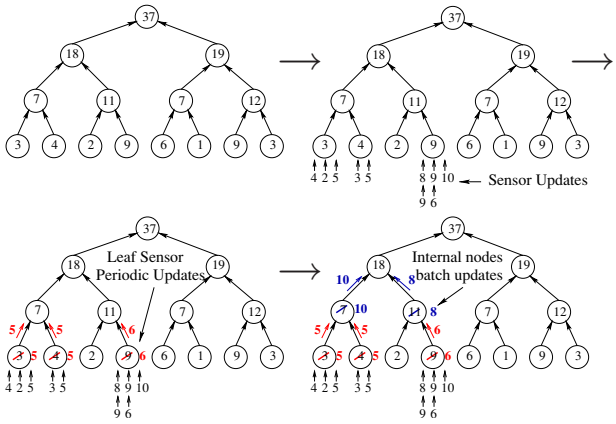


Figure 4: Temporal batching allows leaf sensors to condense a series of updates into a periodic report and allows internal nodes to combine updates from different subtrees before transmitting them further.

system batches a large group of updates together, a short network or node failure can cause a large error. For example, suppose a system is enforcing TI=60s for an attribute, and suppose that an aggregation node near the root has collected 59 seconds worth of updates from its descendents but then loses its connection to the root for a few seconds. That short disruption can cause the system to violate its TI guarantees for a large number of updates.

3 NI Abstraction and Application

To cope with the sources of error just described, we introduce a new consistency metric, Network Imprecision (NI), that addresses the needs of large-scale monitoring systems in environments where networks or nodes can fail.

This section defines NI and argues that it is the right abstraction for a *monitoring system* to provide to *monitoring applications*. The discussions in this section assume that NI is provided by an oracle. Section 4 describes how to compute the NI metrics accurately and efficiently.

3.1 NI metrics

The definition of NI is driven by four fundamental properties of large-scale monitoring systems. First, updates reflect real-world events that are outside of the system's control. Second, updates can occur at large numbers of sensor nodes. Third, systems may support monitoring of large numbers of attributes. Fourth, different applications are affected by and may react to missing updates in different ways.

The first two properties suggest that traditional data consistency algorithms that enforce guarantees like causal consistency [33] or sequential consistency [34] or linearizability [24] are not appropriate for large-scale monitoring systems. To enforce limits on inconsistency, traditional consistency algorithms must block reads or writes during partitions [19]. However, in large-scale monitoring systems (1) updates cannot be blocked because they reflect external events that are not under the system's control and (2) reads depend on inputs at many nodes, so blocking reads when any sensor is unreachable will result in unacceptable availability.

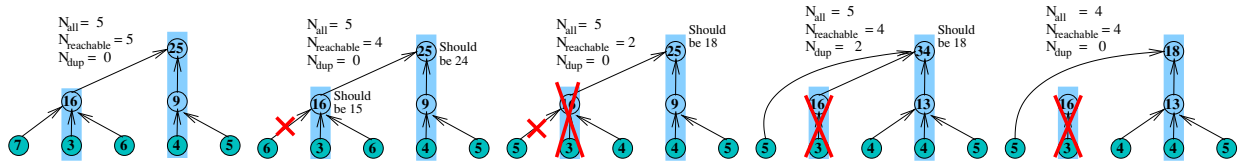
We therefore cast NI as a monitoring system's introspection on its own stability. Rather than attempt to *enforce* limits on the inconsistency of *data items*, a monitoring overlay uses introspection on its current state to produce an NI value that *exposes* the extent to which *system disruptions* may affect results.

In its simplest form, NI could be provided as a simple stability flag. If the system is stable (all nodes are up, all network paths are available, and all updates are propagating within the delays specified by the system's temporal imprecision guarantees), then an application knows that it can trust the monitoring system's outputs. Conversely, if the monitoring system detects that any of these conditions is violated, it could simply flag its outputs as suspect, warning applications that some sensors' updates may not be reflected in the current outputs.

Since large systems may seldom be completely stable and in order to allow different applications sufficient flexibility to handle system disruptions, instead of an all-or-nothing stability flag, our implementation of the NI abstraction quantifies the scope of system disruptions. In particular, we provide three metrics: N_{all}, $N_{reachable}$, and N_{dup}.

- N_{all} estimates the number of nodes in the system.
- $N_{reachable}$ is a lower bound on the number of nodes whose *recent* input values are guaranteed to be reflected in the query result. Recency is defined by the TI guarantees the system provides for the attribute. For example, if the TI is 60 seconds, then $N_{all} - N_{reachable}$ is the number of inputs whose values may be stale by more than 60 seconds.
- N_{dup} provides an upper bound on the number of nodes whose input contribution to an aggregate may be repeated. Repeated inputs can occur when a topology reconfiguration causes a leaf node or a subtree to switch to a new parent while its old parent retains the node's or subtree's input as soft state until a timeout.

These three metrics characterize the consistency of a query result. If $N_{reachable} = N_{all}$ and $N_{dup} = 0$, then query results are *guaranteed* to meet the AI and TI bounds specified by the system. If $N_{reachable}$ is close to N_{all} and N_{dup} is low, the results reflect most inputs and are



(a) Fault-free (b) Node disconnect (c) Amplification (d) Reconfig 1 (e) Reconfig 2

Figure 5: The evolution of $N_{reachable}$, N_{all}, and N_{dup} during failures and reconfigurations. The values in the center of each circle illustrate an example SUM aggregate. The vertical bars show the virtual nodes corresponding to a given physical leaf node.

likely to be useful for many applications. Conversely, query answers with high values of $N_{all} - N_{reachable}$ or N_{dup} suggest that the network is unstable and the results should not be trusted.

Mechanism vs. policy. This formulation of NI explicitly separates the mechanism for network introspection of a monitoring system from application-specific policy for detecting and minimizing the effects of failures, delays, or reconfigurations on query results. Although it is appealing to imagine a system that not only reports how a disruption affects the overlay but also how the disruption affects each monitored attribute, we believe that NI provides the right division of labor between the monitoring system and monitoring applications for three reasons.

First, the impact of omitted or duplicated updates is highly application-dependent, depending on the aggregation function (e.g., some aggregation functions are insensitive to duplicates [12]), the variability of the sensor inputs (e.g., when inputs change slowly, using a cached update for longer than desired may have a modest impact), the nature of the application (e.g., an application that attempts to detect unusual events like network anomalies may reap little value from using statistical techniques for estimating the state of unreachable sensors), and application requirements (e.g., some applications may value availability over correctness and live with best effort answers while others may prefer not to act when the accuracy of information is suspect).

Second, even if it were possible to always estimate the impact of disruptions on applications, hard-wiring the system to do such per-attribute work would impose significant overheads compared to monitoring the status of the overlay.

Third, as we discuss in Section 3.3, there are a broad range of techniques that applications can use to cope with disruptions, and our definition of NI allows each application to select the most appropriate technique.

3.2 Example

Here, we illustrate how NI's three metrics characterize system state using a simple example.

Consider the aggregation tree across 5 physical nodes in Figure 5(a). For simplicity, we compute a SUM aggregate under an AI filtering budget of zero (i.e., update propagation is suppressed if the value of an attribute has not changed), and we assume a TI guarantee of TI_{limit} = 30 seconds (i.e., the system promises a maximum staleness of 30 seconds). Finally, to avoid spurious garbage collection/reconstruction of per-attribute state, the underlying DHT reconfigures its topology if a path is down for a long timeout (e.g., a few minutes), and internal nodes cache inputs from their children as soft state for slightly longer than that amount of time.

Initially, (a) the system is stable; the root reports the correct aggregate value of 25 with $N_{all} = N_{reachable} = 5$ and $N_{dup} = 0$ indicating that all nodes' recent inputs are reflected in the aggregate result with no duplication.

Then, (b) the input value changes from 7 to 6 at a leaf node, but before sending that update, the node gets disconnected from its parent. Because of soft state caching, the failed node's old input is still reflected in the SUM aggregate, but recent changes at that sensor are not; the root reports 25 but the correct answer is 24. As (b) shows, NI exposes this inconsistency to the application by changing $N_{reachable}$ to 4 within TI_{limit} = 30 seconds of the disruption, indicating that the reported result is based on stale information from at most one node.

Next, we show how NI exposes the failure amplification effect. In (c), a single node failure disconnects the entire subtree rooted at that node. NI reveals this major disruption by reducing $N_{reachable}$ to 2 since only two leaves retain a good path to the root. The root still reports 25 but the correct answer (i.e., what an oracle would compute using the live sensors' values as inputs) is 18. Since only 2 of 5 nodes are reachable, this report is suspect. The application can either discard it or take corrective actions such as those discussed in Section 3.3.

NI also exposes the effects of overlay reconfiguration. After a timeout, (d) the affected leaves switch to new parents; NI exposes this change by increasing $N_{reachable}$ to 4. But since the nodes' old values may still be cached, N_{dup} increases to 2 indicating that two nodes' inputs are double counted in the root's answer of 34.

Finally, NI reveals when the system has restabilized.

In (e), the system again reaches a stable state—the soft state expires, N_{dup} falls to zero, N_{all} becomes equal to $N_{reachable}$ of 4, and the root reports the correct aggregate value of 18.

3.3 Using NI

As noted above, NI explicitly separates the problem of characterizing the state of the monitoring system from the problem of assessing how disruptions affect applications. The NI abstraction is nonetheless powerful—it supports a broad range of techniques for coping with network and node disruptions. We first describe four standard techniques we have implemented: (1) flag inconsistent answers, (2) choose the best of several answers, (3) on-demand reaggregation when inconsistency is high, and (4) probing to determine the numerical contribution of duplicate or stale inputs. We then briefly sketch other ways applications can use NI.

Filtering or flagging inconsistent answers. PRISM's first standard technique is to manage the trade-off between consistency and availability [19] by sacrificing availability: applications report an exception rather than returning an answer when the fraction of unreachable or duplicate inputs exceeds a threshold. Alternatively, applications can maximize availability by always returning an answer based on the best available information but flagging that answer's quality as high, medium, or low depending on the number of unreachable or duplicated inputs.

Redundant aggregation. PRISM can aggregate an attribute using k different keys so that one of the keys is likely to find a route around the disruption. Since each key is aggregated using a different tree, each has a different NI associated with it, and the application chooses the result associated with the key that has the best NI. In Section 6, we show that using a small value of k ($k = 4$) reduces the worst-case inaccuracy by nearly a factor of five.

On-demand reaggregation. Given a signal that current results may be affected by significant disruptions, PRISM allows applications to trigger a full on-demand reaggregation to gather current reports (without AI caching or TI buffering) from all available inputs. In particular, if an application receives an answer with unacceptably high fraction of unreachable or duplicated inputs, it issues a probe to force all nodes in the aggregation tree to discard their cached data for the attribute and to recompute the result using the current value at all reachable leaf inputs.

Determine V_{dup} or V_{stale}. When N_{dup} or $N_{all} - N_{reachable}$ is high, an application knows that many inputs may be double counted or stale. An application can gain additional information about how the network disruption affects a specific attribute by computing V_{dup} or V_{stale} for that attribute. V_{dup} is the aggregate function applied to all inputs that indicate that they may also be counted in another subtree; for example in Figure 5(d), V_{dup} is 9 from the two nodes on the left that have taken new parents before they are certain that their old parent's soft state has been reclaimed. Similarly, V_{stale} is the aggregate function applied across cached values from unreachable children; in Figure 5(c) V_{stale} is 16, indicating that 16/25 of the sum value comes from nodes that are currently unreachable.

Since per-attribute V_{dup} and V_{stale} provide more information than the NI metrics, which merely characterize the state of the topology without reference to the aggregation functions or their values, it is natural to ask: Why not always provide V_{dup} and V_{stale} and dispense with the NI metrics entirely? As we will show in Section 4, the NI metrics can be computed efficiently. Conversely, the attribute-specific V_{dup} and V_{stale} metrics must be computed and actively maintained on a per-attribute basis, making them too expensive for monitoring a large number of attributes. Given the range of techniques that can make use of the much cheaper NI metrics, PRISM provides NI as a general mechanism but allows applications that require (and are willing to pay for) the more detailed V_{dup} and V_{stale} information to do so.

Other techniques. For other monitoring applications, it may be useful to apply other domain-specific or application-specific techniques. Examples include

- *Duplicate-insensitive aggregation.* Some applications can be designed with duplicate-insensitive aggregation functions where nodes can transmit copies of aggregate values along different paths to guard against failures without affecting the final result. E.g., MAX is inherently duplicate-insensitive [36], and duplicate-insensitive approximations of some other functions exist [12, 37, 41].

- *Increasing reported TI.* Short bursts of reduced $N_{reachable}$ mean that an aggregated value may not reflect some recent updates. Rather than report a result with low TI staleness but a high NI, the system can report a result with a low NI but an explicitly increased TI staleness bound.

- *Statistical Data Analysis.* Some applications can combine application-level redundancy and statistical inference to estimate missing values, as well as estimating the process parameters for the model generating those values. E.g., Bayesian inference [48] has been used in a one-level tree to estimate missing sensor inputs and model parameters in an environmental sensor network.

These examples are illustrative but not comprehensive. Armed with information about the likely quality of a given answer, applications can take a wide range of approaches to protect themselves from disruptions.

4 Computing NI metrics

The three NI metrics are simple, and implementing them initially seems straightforward: N_{all}, $N_{reachable}$, and N_{dup} are each conceptually aggregates of counts across nodes, which appear to be easy to compute using PRISM's standard aggregation features. However, this simple picture is complicated by two requirements on our solution:

1. *Correctness despite reconfigurations.* PRISM must cope with reconfiguration of dynamically constructed aggregation trees while still guaranteeing the invariants that (a) query results reflect current (to the limits of each attribute's TI bounds) inputs from *at least* $N_{reachable}$ nodes and (b) query results reflect *at most* N_{dup} duplicate inputs due to topology reconfigurations.

2. *Scalability.* PRISM must scale to large numbers of nodes despite (a) the need for active probing to measure liveness between each parent-child pair and (b) the need to compute distinct NI values for each of the distinct aggregation trees in the underlying DHT forest. Naive implementations of NI would incur excessive monitoring overhead as we show in Section 4.3.

In the rest of this section, we first provide a simple algorithm for computing N_{all} and $N_{reachable}$ for a single, static tree. Then, in Section 4.2, we explain how PRISM computes N_{dup} to account for dynamically changing aggregation topologies. Later, in Section 4.3 we describe how to scale the approach to a large number of distinct trees constructed by PRISM's DHT framework.

4.1 Single tree, static topology

This section considers calculating N_{all} and $N_{reachable}$ for a single, static-topology aggregation tree.

N_{all} is simply a count of all nodes in the system, which serves as a baseline for evaluating $N_{reachable}$ and N_{dup}. N_{all} is easily computed using PRISM's aggregation abstraction. Each leaf node inserts 1 to the N_{all} aggregate, which has SUM as its aggregation function.

$N_{reachable}$ for a subtree is a count of the number of leaves that have a *good path* to the root of the subtree, where a good path is a path whose processing and network propagation times currently fall within the system's smallest supported TI bound TI_{min}. The difference $N_{all} - N_{reachable}$ thus represents the number of nodes whose inputs may fail to meet the system's tightest supported staleness bound; we will discuss what happens for attributes with TI bounds larger than TI_{min} momentarily.

Nodes compute $N_{reachable}$ in two steps:

1. *Basic aggregation*: PRISM creates a SUM aggregate and each leaf inserts local value of 1. The root of the tree then gets a count of all nodes.

2. *Aggressive pruning*: $N_{reachable}$ must immediately change if the connection to a subtree is no longer a good path. Therefore, each internal node periodically probes each of its children. If a child c is not responsive, the node removes the subtree c's contribution from the $N_{reachable}$ aggregate and immediately sends the new value up towards the root of the $N_{reachable}$ aggregation tree.

To ensure that $N_{reachable}$ is a lower bound on the number of nodes whose inputs meet their TI bounds, PRISM processes these probes using the same data path in the tree as the standard aggregation processing: a child sends a probe reply only after sending all queued aggregate updates and the parent processes the reply only after processing all previous aggregate updates. As a result, if reliable, FIFO network channels are used, then our algorithm introduces no false negatives: if probes are processed within their timeouts, then so are all aggregate updates. Note that our prototype uses FreePastry [46], which sends updates via unreliable channels, and our experiments in Section 6 do detect a small number of false negatives where a responsive node is counted as reachable even though some recent updates were lost by the network. We also expect few false positives: since probes and updates travel the same path, something that delays processing of probes will likely also affect at least some other attributes.

Supporting temporal batching. If an attribute's TI bound is relaxed to $TI_{attr} > TI_{min}$, PRISM uses the extra time $TI_{attr} - TI_{min}$ to batch updates and reduce load. To implement temporal batching, PRISM defines a narrow window of time during which a node must propagate updates to its parents (assume clocks with bounded drift but not synchronized); details appear in an extended technical report [31]. However, an attribute's subtree that was unreachable over the last TI_{attr} could have been unlucky and missed its window even though it is currently reachable.

To avoid having to calculate a multitude of $N_{reachable}$ values for different TI bounds, PRISM modifies its temporal batching protocol to ensure that each attribute's promised TI bound is met for all nodes counted as reachable. In particular, when a node receives updates from a child marked unreachable, it knows those updates may be late and may have missed their propagation window. It therefore marks such updates as NODELAY. When

a node receives a NODELAY update, it processes the update immediately and propagates the result with the NODELAY flag so that temporal batching is temporarily suspended for that attribute. This modification may send extra messages in the (hopefully) uncommon case of a link performance failure and recovery, but it ensures that $N_{reachable}$ only counts nodes that are meeting all of their TI contracts.

4.2 Dynamic topology

Each virtual node in PRISM caches state from its children so that when a new input from one child arrives, it can use local information to compute new values to pass up. This information is soft state—a parent discards it if a child is unreachable for a long time, similar to IGMP [28].

As a result, when a subtree chooses a new parent, that subtree's inputs may still be stored by a former parent and thus may be counted multiple times in the aggregate as shown in Figure 5(d). N_{dup} exposes this inaccuracy by bounding the number of leaves whose inputs might be included multiple times in the aggregate query result.

The basic aggregation function for N_{dup} is simple: if a subtree root spanning l leaves switches to a new parent, that subtree root inserts the value l into the N_{dup} aggregate, which has SUM as its aggregation function. Later, when sufficient time has elapsed to ensure that the node's old parent has removed its soft state, the node updates its input for the N_{dup} aggregate to 0.

Our N_{dup} implementation must deal with two issues.

1. First, for correctness, we ensure that N_{dup} bounds the number of nodes whose inputs are double counted despite failures and network delays. We ensure this invariant by constructing a hierarchy of leases on a node's right to cache its descendent's soft state such that the leases granted by a node to its parents are always shorter than the leases the node holds from any child whose inputs are reflected in the aggregates maintained by the node.

2. Second, for good performance, we minimize the scope of disruptions when a tree reconfigures using *early expiration*: a node at level i of the tree discards the state of an unresponsive subtree ($maxLevels$ - i) * t_{early} before its lease expires. Early expiration thereby minimizes the scope of a reconfiguration by ensuring that the parent of a failed subtree disconnects that subtree before any higher ancestor is forced to disconnect a larger subtree.

We provide further details on these aspects of the implementation in an extended technical report [31].

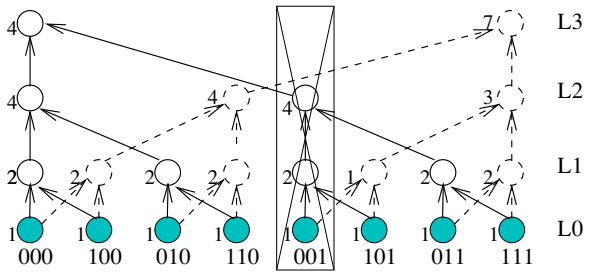


Figure 6: The failure of a physical node has different effects on different aggregations depending on which virtual nodes are mapped to the failed physical node. The numbers next to virtual nodes show the value of $N_{reachable}$ for each subtree after the failure of physical node 001, which acts as a leaf for one tree but as a level-2 subtree root for another.

4.3 Scaling to large systems

Scaling NI is a challenge. To scale attribute monitoring to a large number of nodes and attributes, PRISM constructs a forest of trees using an underlying DHT and then uses different aggregation trees for different attributes [5, 39, 44, 53]. As Figure 6 illustrates, a failure affects different trees differently. The figure shows two aggregation trees corresponding to keys 000 and 111 for an 8-node system. In this system, the failure of the physical node with key 001 removes only a leaf node from the tree 111 but disconnects a 2-level subtree from the tree 000. Therefore, quantifying the effect of failures requires calculating the NI metrics for each of the n distinct trees in an n-node system. Making matters worse, as Section 4.1 explained, maintaining the NI metrics requires frequent active probing along each edge in each tree.

As a result of these factors, the straightforward algorithm for maintaining NI metrics separately for each tree is not tenable: the DHT forest of n degree-d aggregation trees with n physical nodes and each tree having $\frac{n-1/d}{1-1/d}$ edges ($d > 1$), has $\Theta(n^2)$ edges that must be monitored; such monitoring would require $\Theta(n)$ messages per node per probe interval. To put this overhead in perspective, consider a n=1024-node system with d=16-ary trees (i.e., a DHT with 4-bit correction per hop) and a probe interval $p = 10s$. The straightforward algorithm then has each node sending roughly 100 probes per second. As the system grows, the situation deteriorates rapidly—a 16K-node system requires each node to send roughly 1600 probes per second.

Our solution, described below, reduces active monitoring work to $\Theta(\frac{d \log_d n}{p})$ probes per node per second. The 1024-node system in the example would require each node to send about 5 probes per second; the 16K-node system would require each node to send about 7 probes per second.

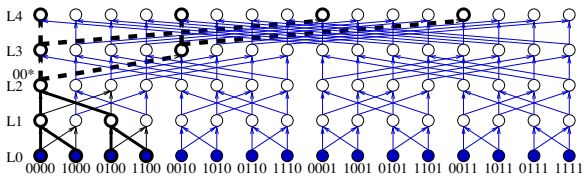


Figure 7: Plaxton tree topology is an approximate butterfly network. Virtual node 00* in a 16-node network uses the dual tree prefix aggregation abstraction to aggregate values from a tree below it (solid bold lines) and distribute the results up a tree above it (dotted bold lines).

Dual tree prefix aggregation. To make it practical to maintain the NI values, we take advantage of the underlying structure of our Plaxton-tree-based DHT [44] to reuse common sub-calculations across different aggregation trees using a novel *dual tree prefix aggregation* abstraction.

As Figure 7 illustrates, this DHT construction forms an approximate butterfly network. For a degree-d tree, the virtual node at level i has an id that matches the keys that it routes in $i * \log d$ bits. It is the root of exactly one tree, and its children are approximately d virtual nodes that match keys in $(i-1) * \log d$ bits. It has d parents, each of which matches different subsets of keys in $(i+1) * \log d$ bits. But notice that for each of these parents, this tree aggregates inputs from *the same subtrees*.

Whereas the standard aggregation abstraction computes a function across a set of subtrees and propagates it to one parent, a *dual tree prefix aggregation* computes an aggregation function across a set of subtrees and propagates it to *all parents*. As Figure 7 illustrates, each node in a dual tree prefix aggregation is the root of two trees: an *aggregation tree* below that computes an aggregation function across nodes in a subtree and a *distribution tree* above that propagates the result of this computation to a collection of enclosing aggregates that depend on this subtree for input.

For example in Figure 7, consider the level 2 virtual node 00* mapped to node 0000. This node's $N_{reachable}$ count of 4 represents the total number of leaves included in that virtual node's subtree. This node aggregates this single $N_{reachable}$ count from its descendants and propagates this value to both of its level-3 parents, 0000 and 0010. For simplicity, the figure shows a binary tree; by default PRISM corrects 4 bits per hop, so each subtree is common to 16 parents.

5 Case-study applications

We have developed a prototype of the PRISM monitoring system on top of FreePastry [46]. To guide the system development and to drive the performance evaluation, we have also built three case-study applications using PRISM: (1) a distributed heavy hitter detection service, (2) a distributed monitoring service for Internet-scale systems, and (3) a distribution detection service for monitoring distributed-denial-of-service (DDoS) attacks at the source-side in large-scale systems.

Distributed Heavy Hitter detection (DHH). Our first application is identifying heavy hitters in a distributed system—for example, the 10 IPs that account for the most incoming traffic in the last 10 minutes [15,30]. The key challenge for this distributed query is scalability for aggregating per-flow statistics for tens of thousands to millions of concurrent flows in real-time. For example, a subset of the Abilene [1] traces used in our experiments include 260 thousand flows that send about 85 million updates in an hour.

To scalably compute the global heavy hitters list, we chain two aggregations where the results from the first feed into the second. First, PRISM calculates the total incoming traffic for each destination address from all nodes in the system using SUM as the aggregation function and hash(HH-Step1, destIP) as the key. For example, tuple (H = hash(HH-Step1, 128.82.121.7), 700 KB) at the root of the aggregation tree T_H indicates that a total of 700 KB of data was received for 128.82.121.7 across all vantage points during the last time window. In the second step, we feed these aggregated total bandwidths for each destination IP into a SELECT-TOP-10 aggregation function with key hash(HH-Step2, TOP-10) to identify the TOP-10 heavy hitters among all flows.

PRISM is the first monitoring system that we are aware of to combine a scalable DHT-based hierarchy, arithmetic filtering, and temporal batching, and this combination dramatically enhances PRISM's ability to support this type of demanding application. To evaluate this application, we use multiple netflow traces obtained from the Abilene [1] backbone network where each router logged per-flow data every 5 minutes, and we replay this trace by splitting it across 400 nodes mapped to 100 Emulab [52] machines. Each node runs PRISM, and DHH application tracks the top 100 flows in terms of bytes received over a 30 second moving window shifted every 10 seconds.

Figure 8(a) shows the precision-performance results as the AI budget is varied from 0% (i.e., suppress an update if no value changes) to 20% of the maximum flow's global traffic volume and as TI is varied from 10 seconds to 5 minutes. We observe that AI of 10% reduces load by an order of magnitude compared to AI of 0 for a fixed TI of 10 seconds, by (a) culling updates for large numbers of "mice" flows whose total bandwidth is less than this value and (b) filtering small changes in the remaining elephant flows. Similarly, TI of 5 minutes reduces load by about 80% compared to TI of 10 seconds. For DHH application, AI filtering is more effective than TI batching for reducing load because of the large fraction of mice flows in the Abilene trace.

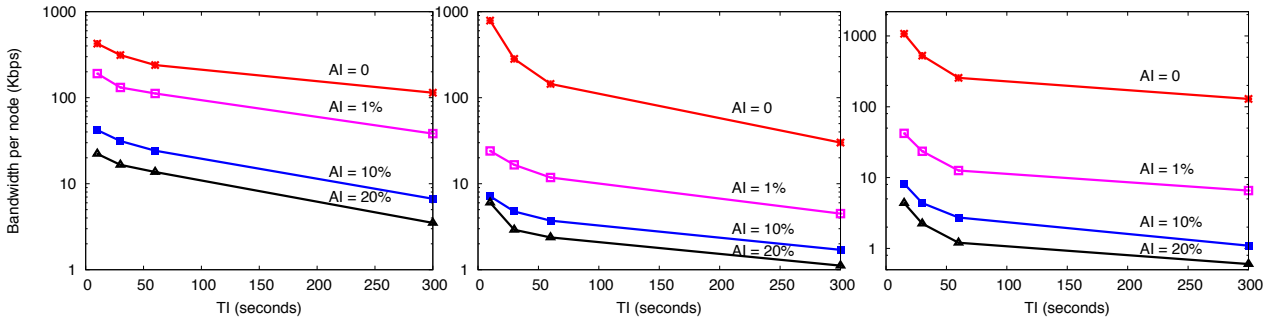


Figure 8: Load vs. AI and TI for (a) DHH, (b) PrMon, and (c) DDoS detection. AI and TI significantly reduce the monitoring load for the three applications; y-axis is on a log scale.

PrMon. The second case-study application is PrMon, a distributed monitoring service that is representative of monitoring Internet-scale systems such as PlanetLab [43] and Grid systems that provide platforms for developing, deploying, and hosting global-scale services. For instance, to manage a wide array of user services running on the PlanetLab testbed, system administrators need a global view of the system to identify problematic services (e.g., any slice consuming more than, say, 10GB of memory across all nodes on which it is running.) Similarly, users require system state information to query for lightly-loaded nodes for deploying new experiments or to track and limit the global resource consumption of their running experiments.

To provide such information in a scalable way and in real-time, PRISM computes the per-slice aggregates for each resource attribute (e.g., CPU, MEM, etc.) along different aggregation trees. This aggregate usage of each slice across all PlanetLab nodes for a given resource attribute (e.g., CPU) is then input to a per-resource SELECT-TOP-100 aggregate (e.g., SELECT-TOP-100, CPU) to compute the list of top-100 slices in terms of consumption of the resource.

We evaluate PrMon using a CoTop [11] trace from 200 PlanetLab [43] nodes at 1-second intervals for 1 hour. The CoTop data provide the per-slice resource usage (e.g., CPU, MEM, etc.) for all slices running on a node. Using these logs as sensor input, we run PrMon on 200 servers mapped to 50 Emulab machines. Figure 8(b) shows the combined effect of AI and TI in reducing PrMon's load for monitoring global resource usage per slice. We observe AI of 1% reduces load by 30x compared to AI of 0 for fixed TI of 10 seconds. Likewise, compared to TI of 10 seconds and AI of 0, TI of 5 minutes reduces overhead per node by 20x. A key benefit of PRISM's tunable precision is the ability to support new, highly-responsive monitoring applications: for approximately the same bandwidth cost as retrieving node state every 5 minutes (TI = 5 minutes, no AI filtering), PRISM provides highly time-responsive and accurate monitoring with TI of 10 seconds and AI of 1%.

DDoS detection at the source. The final monitoring application is DDoS detection to keep track of which nodes are receiving a large number of traffic (bytes, packets) from PlanetLab. This application is important to prevent PlanetLab from being used maliciously or inadvertently to *launch* DDoS traffic (which has, indeed, occurred in the past [2]). For input, we collect a trace of traffic statistics—number of packets sent, number of bytes sent, network protocol, source and destination IP addresses, and source and destination ports—every 15 seconds for four hours using Netfilter's connection tracking interface `/proc/net/ip_conntrack` for all slices from 120 PlanetLab nodes. Each node's traffic statistics are fed into PRISM to compute the aggregate traffic sent to each destination IP across all nodes. Each destination's aggregate value is fed, in turn, to a SELECT-TOP-100 aggregation function to compute a top-100 list of destination IP addresses that receive the highest aggregate traffic (bytes, packets) at two time granularities: (1) a 1 minute sliding window shifted every 15 seconds and (2) a 15 minute sliding window shifted every minute.

Figure 8(c) shows running the application on 120 PRISM nodes mapped to 30 department machines. The AI budget is varied from 0% to 20% of the maximum flow's global traffic volume (bytes, packets) at both the 1 minute and 15 minutes time windows, and TI is varied from 15 seconds to 5 minutes. We observe that AI of 1% reduces load by 30x compared to AI of 0% by filtering most flows that send little traffic. Overall, AI and TI reduce load by up to 100x and 8x, respectively, for this application.

6 Experimental Evaluation

As illustrated above, our initial experience with PRISM is encouraging: PRISM's load-balanced DHT-based hierarchical aggregation, arithmetic filtering, and temporal batching provide excellent scalability and enable demanding new monitoring applications. However, as dis-

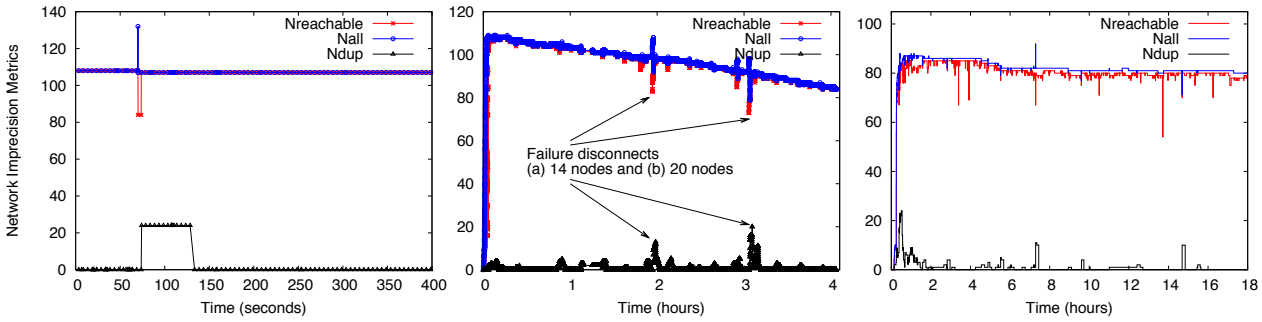


Figure 9: NI metrics under system churn: (a) single node failure at 70 seconds for 108 Emulab nodes, (b) periodic failures for 108 Emulab nodes, and (c) 85 PlanetLab nodes with no synthetic failures.

cussed in Section 2, this scalability comes at a price: the risk that query results depart significantly from reality in the presence of failures.

This section therefore focuses on a simple question: can NI safeguard accuracy in monitoring systems that use hierarchy, arithmetic filtering, or temporal batching for scalability? We first investigate PRISM's ability to use NI to qualify the consistency guarantees promised by AI and TI, then explore the consistency/availability trade-offs that NI exposes, and finally quantify the overhead in computing the NI metrics. Overall, our evaluation shows that NI enables PRISM to be an effective substrate for accurate scalable monitoring: the NI metrics characterize system state and reduce measurement inaccuracy while incurring low communication overheads.

6.1 Exposing disruption

In this section, we illustrate how PRISM uses NI to expose overlay disruptions that could significantly affect monitoring applications.

We first illustrate how NI metrics reflect network state in two controlled experiments in which we run 108 PRISM nodes on Emulab. In Figure 9(a) we kill a single node 70 seconds into the run, which disconnects 24 additional nodes from the aggregation tree being examined. Within seconds, this failure causes $N_{reachable}$ to fall to 83, indicating that any result calculated in this interval might only include the most recent values from 83 nodes. Pastry detects this failure quickly in this case and reconfigures, causing the disconnected nodes to rejoin the tree at a new location. These nodes contribute 24 to N_{dup} until they are certain that their former parent is no longer caching their inputs as soft state. The glitch for N_{all} occurs because the disconnected children rejoin the system slightly more quickly than their prior ancestor detects their departure. Figure 9(b) traces the evolution of the NI metrics as we kill one of the 108 original nodes every 10 minutes over a 4 hour run, and similar behaviors are evident; we use higher churn than typical environments to stress test the system. Figure 9(c) shows how NI reflects network state for a 85-node PrMon experiment on PlanetLab for an 18-hour run; nodes were randomly picked from 248 Internet2 nodes. For some of the following experiments we focus on NI's effectiveness during periods of instability by running experiments on PlanetLab nodes. Because these nodes show heavy load, unexpected delays, and relatively frequent reboots (especially prior to deadlines!), we expect these nodes to exhibit more NI than a typical environment, which makes them a convenient stress test of our system.

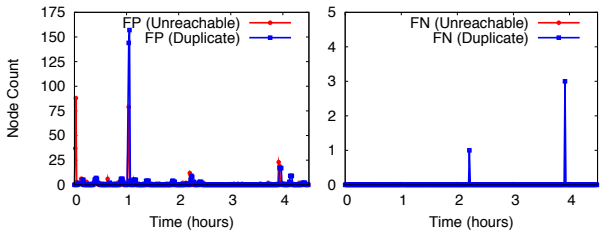
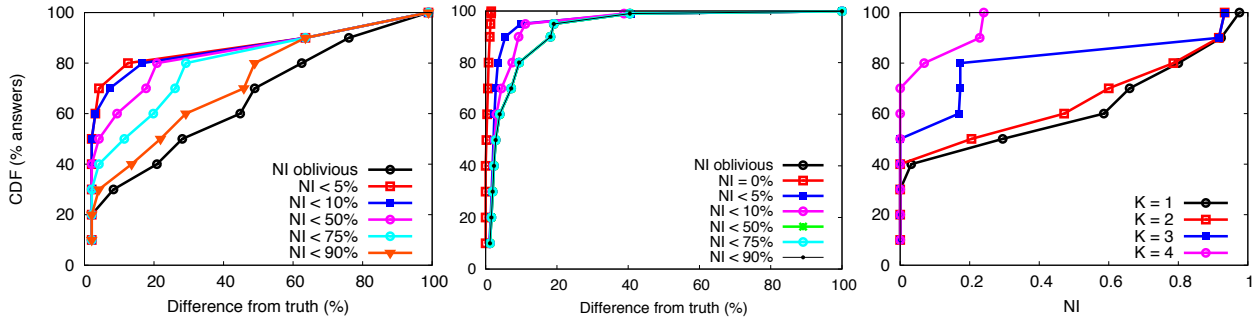


Figure 10: Validation of NI metrics: NI has (a) few false positives and (b) almost zero false negatives.

Our implementation of NI is conservative: we are willing to accept some false positives (when NI reports that inputs are stale or duplicated when they are not, as discussed in Section 4.1), but we want to minimize false negatives (when NI fails to warn an application of duplicate or stale inputs). Figure 10 shows the results for a 96-node Emulab setup under a network path failure model [13]; we use failure probability, MTTF, and MTTR of 0.1, 3.2 hours, and 10 minutes. We contrive an "append" aggregation in each of the 96 distinct trees: each node periodically feeds a (node, timestamp) tuple and internal nodes append the children inputs together. At each root, we compare the aggregate value against NI to detect any false positive (FP) or false negative (FN) reports for (1) N_{dup} count duplicates and (2) dips in $N_{reachable}$ count nodes whose reported values are not within TI of



(a) CDF of result accuracy (PlanetLab) (b) CDF of result accuracy (Emulab) (c) CDF of observed NI (PlanetLab)

Figure 11: (a) and (b) show the CDFs of result accuracy with answers filtered for different NI thresholds and k = 1 for PrMon on (a) Planetlab and (b) Emulab. (c) Shows the availability of results on Planetlab by showing the CDF of NI values for k duplicate keys.

their current values.

Figure 10 indicates that NI accurately characterizes how churn affects aggregation. Across 300 minutes and 96 trees, we observe fewer than 100 false positive reports, most of them small in magnitude. The high FP near 60 minutes is due to a root failure triggering a large reconfiguration and was validated using logs. We observe zero false negative reports for unreachability and three false negative reports for duplication; the largest error was underreporting the number of duplicate nodes by three. Overall, we observe a FP rate less than 0.3% and a FN rate less than 0.01%.

6.2 Coping with disruption

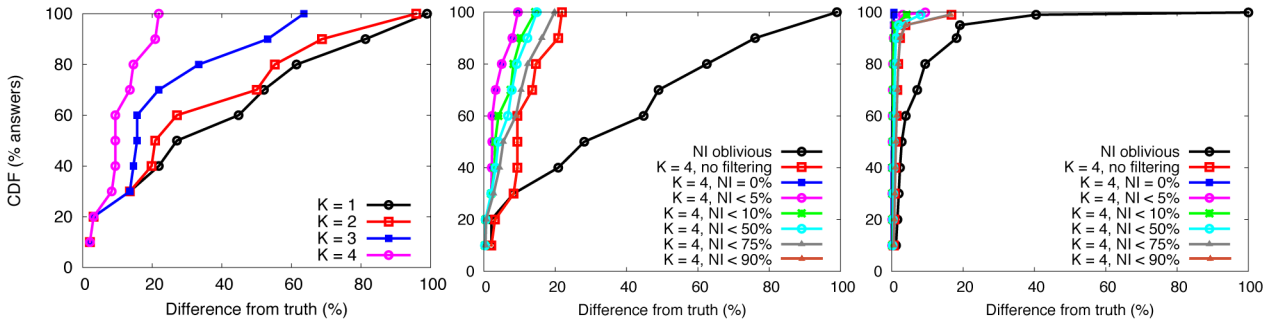
We now examine how applications use NI to improve their accuracy by compensating for churn. Our basic approach is to compare results of NI-oblivious aggregation and aggregation with NI-based compensation with an oracle that has access to the inputs at all leaves; we simulate the oracle via off-line processing of input logs. We run a 1 hour trace-based PrMon experiment on 94 PlanetLab nodes or 108 Emulab nodes for a collection of attributes calculated using a SUM aggregate with AI = 0 and TI = 60 seconds. For Emulab, we use the synthetic failure model described for Fig 10. Note that to keep the discussion simple, we condense NI to a single parameter: NI = $\left(\frac{N_{all}-N_{reachable}}{N_{all}}\right)$ + $\left(\frac{N_{dup}}{N_{all}}\right)$ for all the subsequent experiments.

The *NI-oblivious* line of Figure 11(a) and (b) shows for PrMon nodes that ignore NI, the CDF of the difference between query results and the true value of the aggregation computed by an off-line oracle from traces. For PlanetLab, 50% of the reports differ from the truth by more than 30% in this challenging environment. For the more stable Emulab environment, a few results differ from reality by more than 40%. Next, we discuss how applications can achieve better accuracy using techniques discussed in Section 3.3.

Filtering. One technique is to trade availability for accuracy by filtering results during periods of instability. The lines (NI $< x$%, x = 5, 10, 50, 75, and 90) of Figure 11(a) and (b) show how rejecting results with high NI improves accuracy. For example, for the high-churn PlanetLab environment, when NI $<$ 5%, 80% answers have less than 20% deviation from the true value. For the Emulab experiment, 95% answers have less than 20% deviation using NI $<$ 5% filtering.

Filtering answers during periods of high churn exposes a fundamental consistency versus availability tradeoff [19]. Applications must decide whether it is better to silently give a potentially inaccurate answer or explicitly indicate when it cannot provide a good answer. For example, the $k = 1$ line in Figure 11(c) shows the CDF of the fraction of time for which NI is at or below a specified value for the PlanetLab run. For half of the reports, NI $>$ 30% and for 20% of the reports, NI $>$ 80% reflecting high system instability. Note that the PlanetLab environment is intended to illustrate PRISM's behavior during intervals of high churn. Since accuracy is maximized when answers reflect complete and current information, systems with fewer disruptions (e.g., Emulab) are expected to show higher result accuracy compared to PlanetLab and we observe this behavior for Emulab where the curves in Figure 11(c) shift up and to the left (graph omitted due to space constraints; please see the technical report [31]).

Redundant aggregation. Redundant aggregation allows applications to trade increased overheads for better availability and accuracy. Rather than aggregating an attribute up a single tree, information can be aggregated up k distinct trees. As k increases, the fraction of time during which NI is low increases. Because the vast majority of nodes in a 16-ary tree are near the leaves, sampling several trees rapidly increases the probability that at least one tree avoids encountering many near-root failures. We provide an analytic model formalizing this intuition in a technical report [31].



(a) PlanetLab redun. aggr. (b) PlanetLab redun. aggr. + filtering (c) Emulab redun. aggr. + filtering

Figure 12: CDF of result accuracy for redundant aggregation up k trees and filtering for PlanetLab and Emulab runs for the PrMon application.

In Figure 12(a) we explore the effectiveness of a simple redundant aggregation approach in which PrMon aggregates each attribute k times and then chooses the result with the lowest NI. This approach maximizes availability—as long as any of the root nodes for an attribute are available, PrMon always returns a result—and it also can achieve high accuracy. Due to space constraints, we focus on the PlanetLab run and show the CDF of results with respect to the deviation from an oracle as we vary k from 1 to 4. We observe that this approach can reduce the deviation to at most 22% thereby reducing the worst-case inaccuracy by nearly 5x.

Applications can combine the redundant aggregation and filtering techniques to get excellent availability and accuracy. Figure 12(b) and (c) show the results for the PlanetLab and Emulab environments. As Figure 11(c) shows, redundant aggregation increases availability by increasing the fraction of time NI is below the filter threshold, and as Figures 12(b) and (c) show, the combination improves accuracy by up to an order of magnitude over best effort results.

6.3 NI scalability

Finally, we quantify the monitoring overhead of tracking NI via (1) each aggregation tree separately and (2) dual-tree prefix aggregation. Figure 13 shows the average per-node message cost for NI monitoring as we vary the network size from 16 to 1024 nodes mapped to 256 Lonestar [35] machines. We observe that the overhead using independent aggregation trees scales linearly with the network size whereas it scales logarithmically using dual-tree prefix aggregation.

Note that the above experiment constructs all n distinct trees in the DHT forest of n nodes assuming that the number of attributes is at least on the order of the number of nodes n. However, for systems that aggregate fewer attributes (or if only few attributes care about NI), it is important to know which of the two techniques for tracking NI—(1) per-tree aggregation or (2) dual-tree prefix aggregation—is more efficient. Figure 14 shows both the average and the maximum message cost across all nodes in a 1000-node experimental setup as above for both per-tree NI aggregation and dual-tree prefix aggregation as we increase the number of trees along which NI value is computed. Note that per-tree NI aggregation costs increase as we increase the number of trees while dual-tree prefix aggregation has a constant cost. We observe that the break-even point for the average load is 44 trees while the break-even point for the maximum load is only 8 trees.

7 Related Work

PRISM is a monitoring architecture that is to our knowledge the first to maximize scalability by integrating three techniques that have been used in isolation in prior systems: DHT-based hierarchy for load balancing and in-network filtering [5,44,47,53], arithmetic filtering [6,21, 30, 31, 36, 38, 42, 51, 56], and temporal batching [14, 36, 56]. As discussed in Section 2, each of these techniques improves scalability, but each also increases the risk that queries will report incorrect results during network and node failures. We believe the NI abstraction and the implementation techniques discussed in this paper will be widely applicable.

The idea of flagging results when the state of a distributed system is disrupted by node or network failures has been used in tackling other distributed systems problems. For example, our idea of NI is analogous to that of fail-aware services [16] and failure detectors [9] for fault-tolerant distributed systems. Freedman et al. propose link-attestation groups [18] that use an application specific notion of reliability and correctness to map pairs of nodes which consider each other reliable. Their system, designed for groups on the scale of tens of nodes, monitors the nodes and system and exposes such attestation graph to the applications. Bawa et al. [4] survey previous work on measuring the validity of query results in faulty networks. Their "single-site validity" semantic is equiv-

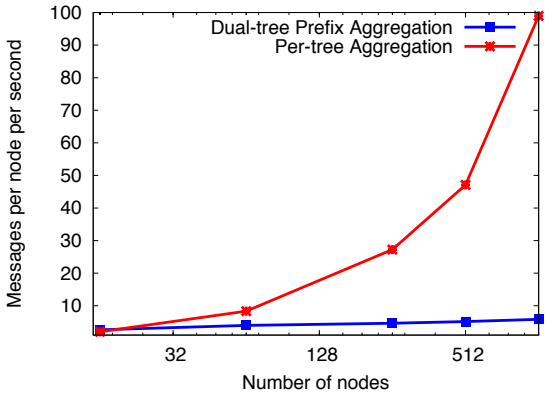


Figure 13: NI monitoring overhead for dual-tree prefix aggregation and for computing NI per aggregation tree. The overhead scales linearly with the network size for per-tree aggregation whereas it scales logarithmically using dual-tree prefix aggregation.

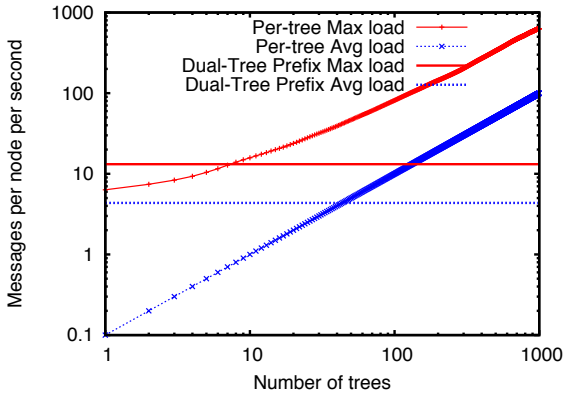


Figure 14: Break-even point for NI tracking overhead as the number of trees (attributes) varies for (a) per-tree aggregation vs. (b) dual-tree prefix aggregation in a 1000-node system. The break-even points for the average and maximum load are 44 trees and 8 trees.

alent to PRISM's $N_{reachable}$ metric. Completeness [20] defined as the percentage of network hosts whose data contributed to the final query result, is similar to the ratio of $N_{reachable}$ and N_{all}. Relative Error [12, 57] between the reported and the "true" result at any instant can only be computed by an oracle with a perfect view of the dynamic network.

Several aggregation systems have worked to address the failure amplification effect. To mask failures, TAG [36] proposes (1) reusing previously cached values and (2) dividing the aggregate value into fractions equal to the number of parents and then sending each fraction to a distinct parent. This approach reduces the variance but not the expected error of the aggregate value at the root. SAAR uses multiple interior-node-disjoint trees to reduce the impact of node failures [39]. In San Fermin [8], each node creates its own binomial tree by swapping data with other nodes. Seaweed [40] uses a supernode approach in which data on each internal node is replicated. However, both these systems process one-shot queries but not continuous queries on high-volume dynamic data, which is the focus of PRISM. Gossip-based protocols [7, 45, 51] are highly robust but incur more overhead than trees [53]. NI can also complement gossip protocols, which we leave as future work. Other studies have proposed multi-path routing methods [12,20,29,37, 41] for fault-tolerant aggregation.

Recent proposals [4,12,37,41,55] have combined multipath routing with order- and duplicate-insensitive data structures to tolerate faults in sensor network aggregation. The key idea is to use probabilistic counting [17] to approximately count the number of distinct elements in a multi-set. PRISM takes a complementary approach: whereas multipath duplicate-insensitive (MDI) aggregation seeks to reduce the effects of network disruption, PRISM's NI metric seeks to quantify the network disruptions that do occur. In particular, although MDI aggregation can, in principle, reduce network-induced inaccuracy to any desired target if losses are independent and sufficient redundant transmissions are made [41], the systems studied in the literature are still subject to non-zero network-induced inaccuracy due to efforts to balance transmission overhead with loss rates, insufficient redundancy in a topology to meet desired path redundancy, or correlated network losses across multiple links. These issues may be more severe in our environment than in wireless sensor networks targeted by MDI approaches because the dominant loss model may differ (e.g., link congestion and DHT reconfigurations in our environment versus distance-sensitive loss probability for the wireless sensors) and because the transmission cost model differs (for some wireless networks, transmission to multiple destinations can be accomplished with a single broadcast.

The MDI aggregation techniques are also complementary in that PRISM's infrastructure provides NI information that is common across attributes while the MDI approach modifies the computation of individual attributes. As Section 3.3 discussed, NI provides a basis for integrating a broad range of techniques for coping with network error, and MDI aggregation may be a useful technique in cases when (a) an aggregation function can be recast to be order- and duplicate-insensitive and (b) the system is willing to pay the extra network cost to transmit each attribute's updates. To realize this promise, additional work is required to extend MDI approaches for bounding the approximation error while still minimizing network load via AI and TI filtering.

8 Conclusions

> If a man will begin with certainties, he shall end in doubts; but if he will be content to begin with doubts, he shall end in certainties.
> –Sir Francis Bacon

We have presented Network Imprecision, a new metric for characterizing network state that quantifies the consistency of query results in a dynamic, large-scale monitoring system. Without NI guarantees, large scale network monitoring systems may provide misleading reports because query result outputs by such systems may be arbitrarily wrong. Incorporating NI in the PRISM monitoring framework qualitatively improves its output by exposing cases when approximation bounds on query results can not be trusted.

9 Acknowledgments

We thank our shepherd Dejan Kostic, Joe Hellerstein, and the anonymous reviewers for their valuable feedback. Navendu Jain is supported by an IBM Ph.D. Fellowship. This work is supported in part by NSF Awards CNS-0546720, CNS-0627020 and SCI-0438314.

References

[1] `http://abilene.internet2.edu/`.

[2] R. Adams. Distributed system management: PlanetLab incidents and management tools. Technical Report PDN–03–015, PlanetLab Consortium, 2003.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[4] M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani. The price of validity in dynamic networks. In *SIGMOD*, 2004.

[5] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *SIGCOMM*, 2004.

[6] M. Bhide, K. Ramamritham, and M. Agrawal. Efficient execution of continuous incoherency bounded queries over multi-source streaming data. In *ICDCS*, 2007.

[7] Y. Birk, I. Keidar, L. Liss, and A. Schuster. Efficient dynamic aggregation. In *DISC*, 2006.

[8] J. Cappos and J. H. Hartman. San fermín: aggregating large data sets using a binomial swap forest. In *NSDI*, 2008.

[9] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 1996.

[10] D. D. Clark, C. Partridge, J. C. Ramming, and J. Wroclawski. A knowledge plane for the Internet. In *SIGCOMM*, 2003.

[11] `http://comon.cs.princeton.edu/`.

[12] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. In *ICDE*, 2004.

[13] M. Dahlin, B. Chandra, L. Gao, and A. Nayate. End-to-end WAN service availability. *IEEE/ACM Transactions on Networking*, 2003.

[14] A. Deshpande, S. Nath, P. Gibbons, and S. Seshan. Cache-and-query for wide area sensor databases. In *SIGMOD*, 2003.

[15] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *SIGCOMM*, 2002.

[16] C. Fetzer and F. Cristian. Fail-awareness in timed asynchronous systems. In *PODC*, 1996.

[17] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *JCSS*, 1985.

[18] M. J. Freedman, I. Stoica, D. Mazieres, and S. Shenker. Group therapy for systems: Using link attestations to manage failures. In *IPTPS*, 2006.

[19] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of Consistent, Available, Partition-tolerant web services. In *ACM SIGACT News, 33(2)*, Jun 2002.

[20] I. Gupta, R. van Renesse, and K. P. Birman. Scalable fault-tolerant aggregation in large process groups. In *DSN*, 2001.

[21] R. Gupta and K. Ramamritham. Optimized query planning of continuous aggregation queries in dynamic data dissemination networks. In *WWW*, pages 321–330, 2007.

[22] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *USITS*, March 2003.

[23] J. M. Hellerstein, V. Paxson, L. L. Peterson, T. Roscoe, S. Shenker, and D. Wetherall. The network oracle. *IEEE Data Eng. Bull.*, 2005.

[24] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Sys.*, 12(3), 1990.

[25] E. Hoke, J. Sun, J. D. Strunk, G. R. Ganger, and C. Faloutsos. Intemon: continuous mining of sensor data in large-scale self-infrastructures. *Operating Systems Review*, 40(3):38–44, 2006.

[26] L. Huang, M. Garofalakis, A. D. Joseph, and N. Taft. Communication-efficient tracking of distributed cumulative triggers. In *ICDCS*, 2007.

[27] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *VLDB*, 2003.

[28] `http://www.ietf.org/rfc/rfc2236.txt`.

[29] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MobiCom*, 2000.

[30] N. Jain, D. Kit, P. Mahajan, P. Yalagandula, M. Dahlin, and Y. Zhang. STAR: Self tuning aggregation for scalable monitoring. In *VLDB*, 2007.

[31] N. Jain, P. Mahajan, D. Kit, P. Yalagandula, M. Dahlin, and Y. Zhang. Network Imprecision: A new consistency

metric for scalable monitoring (extended). Technical Report TR-08-40, UT Austin Department of Computer Sciences, October 2008.

[32] N. Jain, P. Yalagandula, M. Dahlin, and Y. Zhang. Self-tuning, bandwidth-aware monitoring for dynamic data streams. In *ICDE*, 2009.

[33] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7), July 1978.

[34] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.

[35] `http://www.tacc.utexas.edu/resources/hpcsystems`.

[36] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *OSDI*, 2002.

[37] A. Manjhi, S. Nath, and P. B. Gibbons. Tributaries and deltas: efficient and robust aggregation in sensor network streams. In *SIGMOD*, 2005.

[38] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (recently) frequent items in distributed data streams. In *ICDE*, 2005.

[39] A. Nandi, A. Ganjam, P. Druschel, T. S. E. Ng, I. Stoica, H. Zhang, and B. Bhattacharjee. SAAR: A shared control plane for overlay multicast. In *NSDI*, 2007.

[40] D. Narayanan, A. Donnelly, R. Mortier, and A. I. T. Rowstron. Delay aware querying with seaweed. In *VLDB*, 2006.

[41] S. Nath, P. B. Gibbons, S. Seshan, and Z. R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *SenSys*, 2004.

[42] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD*, 2003.

[43] Planetlab. `http://www.planet-lab.org`.

[44] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *ACM SPAA*, 1997.

[45] B. Raghavan, K. V. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren. Cloud control with distributed rate limiting. In *SIGCOMM*, 2007.

[46] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-peer Systems. In *Middleware*, 2001.

[47] J. Shneidman, P. Pietzuch, J. Ledlie, M. Roussopoulos, M. Seltzer, and M. Welsh. Hourglass: An Infrastructure for Connecting Sensor Networks and Applications. Technical Report TR-21-04, Harvard University, 2004.

[48] A. Silberstein, G. Puggioni, A. Gelfand, K. Munagala, and J. Yang. Suppression and failures in sensor networks: A Bayesian approach. In *VLDB*, 2007.

[49] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *ACM SIGCOMM*, 2001.

[50] E. Thereska, B. Salmon, J. D. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: tracking activity in a distributed storage system. In *SIGMETRICS/Performance*, pages 3–14, 2006.

[51] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *TOCS*, 21(2):164–206, 2003.

[52] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, 2002.

[53] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *SIGCOMM*, Aug. 2004.

[54] P. Yalagandula, P. Sharma, S. Banerjee, S.-J. Lee, and S. Basu. S^3: A Scalable Sensing Service for Monitoring Large Networked Systems. In *SIGCOMM Wkshp. on Internet Network Mgmt*, 2006.

[55] H. Yu. Dos-resilient secure aggregation queries in sensor networks. In *PODC*, pages 394–395, 2007.

[56] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *TOCS*, 2002.

[57] Y. Zhao, R. Govindan, and D. Estrin. Computing aggregates for monitoring wireless sensor networks. In *SNPA*, 2003.

Lightweight, High-Resolution Monitoring for Troubleshooting Production Systems

Sapan Bhatia, *Princeton University*
Abhishek Kumar, *Google Inc.*
Marc E. Fiuczynski, *Princeton University*
Larry Peterson, *Princeton University*

Abstract

Production systems are commonly plagued by intermittent problems that are difficult to diagnose. This paper describes a new diagnostic tool, called Chopstix, that continuously collects profiles of low-level OS events (e.g., scheduling, L2 cache misses, CPU utilization, I/O operations, page allocation, locking) at the granularity of executables, procedures and instructions. Chopstix then reconstructs these events offline for analysis. We have used Chopstix to diagnose several elusive problems in a large-scale production system, thereby reducing these intermittent problems to reproducible bugs that can be debugged using standard techniques. The key to Chopstix is an approximate data collection strategy that incurs very low overhead. An evaluation shows Chopstix requires under 1% of the CPU, under 256KB of RAM, and under 16MB of disk space per day to collect a rich set of system-wide data.

1 Introduction

Troubleshooting complex software systems is notoriously difficult. Programmers employ a wide array of tools to help harden their code before it is deployed—ranging from profilers [1, 10, 11, 12, 30] to model checkers [29] to test suites [2]—but fully protecting production systems from undetected software bugs, intermittent hardware failures, configuration errors, and unanticipated workloads is problematic.

Our experience operating PlanetLab [22] convinces us that no amount of pre-production (*offline*) testing will uncover all the behaviors a built-from-components system will exhibit when stressed by production (*online*) workloads. Unfortunately, it is also our experience that existing online tools do not offer sufficient information to diagnose the root cause of such problems, and in many cases, it is difficult to even narrow in on a descriptive set of symptoms. In short, we have found production-only problems hard to reproduce and diagnose using existing tools and techniques. We attribute this situation to three unique challenges:

- Enough data must be available to not only answer the binary question "is the system healthy" but to also understand why or why not. This data should not be restricted to a set of detectable failures (e.g. access violations) but should enable the diagnosis of faults that cannot be characterized *a priori*. The data must be collected at all times, to capture root causes temporally distant from faults and failures. It must also be collected for the whole system to capture "domino effects" that are typical in built-from-components systems.
- Diagnosis must be possible even in conditions in which the system crashes, preventing postmortem analysis of the live system. It should not be restricted to deterministic problems and it should be able to handle complex scenarios that involve non-determinism (e.g., disk-I/O or network packet timing).
- System monitoring and diagnosis must not impose prohibitive overhead, depend on modifications to applications or the OS, or require taking the system offline.

To address these challenges, we developed Chopstix, a tool that continuously collects succinct summaries of the system's behavior, spanning both the OS and applications. These summaries contain comprehensive (with a coverage of 99.9%) logs of events that cut across low-level OS operations (e.g., page allocation, process scheduling, locking and block-level I/O), storing detailed contextual information (e.g., stack traces) for each event.

Chopstix then aggregates and visualizes this data to reconstruct complex misbehaviors. For example, unresponsive programs can be investigated *quantitatively* by analyzing total scheduling delays, which may be due to resource blocking or other reasons. Relevant events (i.e., delayed schedulings of the program) can then be studied *qualitatively* to isolate their root cause (e.g., by studying partial stack traces collected with the event).

It is our contention that Chopstix hits a sweet spot in the online diagnostic toolkit design space in that it is:

- *Comprehensive*. The events tracked have a high-frequency (e.g., once per context switch), which en-

ables them to capture deviations in the system's behavior even when they are short-lived. Each event sample is sufficiently detailed so that its circumstances can be accurately reconstructed.

- *Robust*. It covers the whole system and maintains a long-term execution history, currently measured in weeks. This history can be used to analyze crashes and to interpret complex and unforeseen system behavior.

- *Unintrusive*. It has negligible run-time overhead and space requirements: typically $< 1\%$ CPU utilization, 256Kbytes RAM and 16MB of daily log.

The combination of these features has made Chopstix suitable to help debug a class of problems that are difficult to diagnose by construction, problems that are intermittent and cannot be characterized. For example, it offers sufficient coverage to answer a question such as "what was the system doing last Wednesday around 5pm when the ssh prompt latency was temporarily high, yet system load appeared to be low?" In doing so, it is comprehensive enough to pinpoint the function calls where the system was blocked, combined with (partial) stack traces that may lead to the potential root causes of such a problem. Specifically, Chopstix's value is to help isolate the root cause of a fault or anomaly in sufficient detail to enable the responsible developer to localize and reproduce it—and hence, fully debug it using existing debugging tools.

In describing Chopstix, this paper makes contributions at two levels. At the mechanistic level, it describes how a probabilistic data structure—called *sketches*—enables data collection that is both detailed and lightweight. At the semantic level, it shows how system administrators can use this data to isolate problems arising from non-obvious behavior inherent in complex computer systems. On the latter point, the paper narrates our experiences diagnosing stubborn bugs encountered in PlanetLab.

2 System Overview

This section gives a brief overview of Chopstix, providing enough context for the usage scenarios presented in the next Section. Chopstix consists of three components: a *data collector*, a *data aggregator*, and a *visualizer*. The collector is implemented in the kernel of each monitored node and controlled by a companion user process that periodically copies data from the kernel to the disk. A polling process regularly fetches this data from various nodes to a central location. The aggregator processes this data at multiple timescales (5 minutes, 1 hour, 1 day) and filters it with a set of user-defined aggregation functions. The output of these functions, along with overall trends in the raw data, are rendered in the visualizer and can be viewed—using a conventional web browser—for each timescale.

Although each of these components is integral to the overall workflow, the Chopstix data collector represents the key contribution—the ability to collect extensive monitoring data at low overhead. This section presents the salient features of the Chopstix collector and briefly summarizes the role of the aggregator and visualizer.

2.1 Data Collector

Monitoring tools for production systems must function within a limited set of resources. Chopstix addresses the tradeoff between monitoring visibility and monitoring overhead by introducing *sketches* [14, 15] to the domain of system monitoring.

A sketch is a probabilistic data structure that allows the *approximate* tracking of a large number of events at the same cost as deterministically tracking significantly fewer events. Sketches are an alternative to *uniform sampling*, which has the disadvantage of drawing most of its samples from events with large populations.

To see the limitation of uniform sampling, imagine two processes that are scheduled 99% of the time and 1% of the time, respectively. For a scheduler clock rate of 1000Hz, a uniform sampling rate of 1/100 would require the latter process run for a period of $\approx$7 seconds for the probability that it be recorded to be over 50%. For shorter periods, it is unlikely to be sampled. Sketches cope with this situation by setting the sampling rate for a given event to be a decreasing function of its current population in a given monitoring interval. As a result, events with small and medium-sized populations that would otherwise "fly under the radar" are more likely to be counted.

Specifically, a sketch treats an event in two steps. In the first step, it updates its approximation of the event's population in the current monitoring interval. In the second step, it evaluates a sampling function to determine if this event should be sampled, and if so, saves a sample of relevant information. Specific details on the choice of the data structure and sampling functions used are provided in Section 4 and their error rate is evaluated in Section 5. For now, we concentrate on the use of this data structure.

Sketches make it possible to continuously track a variety of generic system events—e.g., process scheduling, mutex locking, page allocation—whose frequency would overwhelm the resources available for monitoring if they were tracked deterministically. Accordingly, event populations corresponding to each such event type are stored in their own sketch. Section 4 describes this data structure in more detail; for now, we focus on event types, which

we sometimes call the system's *vital signs*.[1]

An event is given by a tuple that includes its type (i.e., the vital sign to which it belongs), a virtual address (e.g., the value of the program counter at the time the event occurs), an executable-file identifier, a user identifier, and a weight (this field is used by the sketch data structure). The tuple also includes a stack trace and other event-specific variables that depict the circumstances of the event. For example, a scheduling-delay event specifies that a process—identified as a username/program pair—was not scheduled in spite of being runnable for a period corresponding to the event's magnitude; the event includes a stack trace taken as the process waited.

Using Chopstix to monitor nodes in PlanetLab as well as other systems running open-source software has helped refine our notions of what vital signs should be chosen as part of the monitoring set. The kinds of bugs we see typically have temporal and spatial ambiguities. These bugs typically cannot be reproduced, and when they happen, they do not leave behind a trail of evidence. Since we begin without *a priori* knowledge about the problem, or even a complete characterization of the problem, we devise a strategy to study deviations in the health of the system. Thus, a good vital sign for monitoring is a property that serves as a useful witness—has a strong temporal and spacial presence—so that when incidents do occur, it captures enough information to reconstruct the system's behavior.

To this end, Chopstix monitors 11 vital signs, chosen to cut across OS operations at the lowest level. The current set includes: CPU utilization, scheduling delay, resource blocking, disk I/O activity, user page allocation, kernel page allocation, L2-cache utilization, system call invocation, signal delivery, socket transmission, and mutex/semaphore locking. Note that we do not claim this set is complete, but the set is easily extended by adding new collectors statically (through code patches for the kernel), or dynamically (through loadable modules or an instrumentation toolkit [20, 3]). Capturing a new vital sign can be accomplished by deploying as little as six lines of code.

Continuing the metaphor of computer system as medical patient, we treat unusual behavior in one or more vital signs as a *symptom* of the problem we are diagnosing. This definition is admittedly vague, where identifying and interpreting symptoms corresponds to the art of diagnosing a problem. We return to this process in Section 3.

For now, we give a simple example. A steady overall rate of disk I/O might indicate good health of the I/O subsystem. However, observing breakdowns of this rate—e.g., specific I/O requests and responses—may reveal requests that are not satisfied (i.e., the request size was greater than the response size). Such behavior would contradict the assumption that all requests are satisfied, indicating a disk problem is possible.

[1] We have found the parallels between medical diagnosis and troubleshooting complex software systems amazingly rich, and so we have adopted terminology from the former when the intuition they provide is helpful.

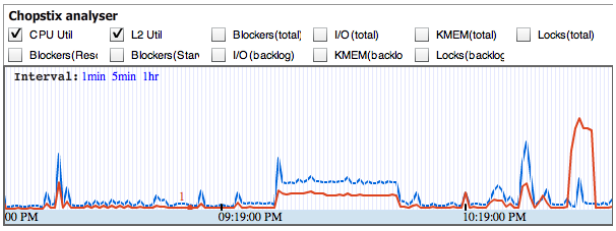


Figure 1: A screenshot of the Chopstix visualizer. Two vital signs are visualized together: CPU utilization (red – the smooth line) and L2-cache misses (blue – the "spiky" line). The X-axis corresponds to time and the Y-axis is normalized. Selecting an epoch, timescale and any vital (not necessarily the ones visualized) displays filtered samples for that vital. This information is not shown in the figure.

2.2 Data Aggregator

The data collected by Chopstix on a node is periodically transferred to a central location where it is processed and rendered. Data from one or more nodes is aggregated by an engine tailored to time-series multi-sets. Specifically, the raw data is aggregated at fixed-sized timescales of 5 minutes, 1 hour and 1 day and passed to a set of multi-set aggregation operators (called filters) via a well-defined API that can be used to deploy new operators. For example, an experimental Exponentially-Weighted-Moving-Average (EWMA) anomaly detector has been implemented as such an operator in about 80 lines of code. Other similar functions sort the sets, measure affinities between items across epochs, and so on. The aggregator also reconstructs the sketching data structures to compute the values of the vital signs.

2.3 Visualizer

The visualizer accepts user requests for specific time ranges and passes them on to the aggregator, which returns the vital signs corresponding to each epoch in the range. These statistics are plotted as a line graph. A user can compare vitals and look for symptoms by overlapping graphs and zooming into specific epochs of interest to retrieve detailed samples, including breakdowns of the vital signs. Clicking on a sample brings up further event-specific information such as a stack trace.

Figure 1 presents a screenshot of the visualizer, showing the main *navigator* window in which the vital signs of interest can be selected and overlapped. The X-axis corresponds to time, with a least count of 1 minute, and the Y-axis is normalized. Specific epochs can be clicked on at a

given timescale to view filtered samples for any vital sign. In this example, the spike at the far right was detected to be a busy loop in the web browser Opera. The correlation that led to this conjecture is a high CPU-utilization rate at a comparatively low L2-cache utilization rate, which we have observed to be a feature of busy loops.

3 Usage Model

This section traces the Chopstix workflow, using examples drawn from our experiences diagnosing problems on PlanetLab to illustrate the insights into system behavior revealed by Chopstix.

One obvious takeaway from this section is that diagnosing system failures is a difficult and inherently human-intensive activity. Re-enforcing that conclusion is not the point of this section. The main contribution is to demonstrate how detailed information collected on a production system can effectively be used to isolate the most stubborn bugs. A secondary contribution is to summarize several "system diagnosing rules" that we believe have general applicability.

3.1 Workflow

The basic workflow is as follows. A user decides to use Chopstix to diagnose a misbehavior that either has temporal ambiguity (a bug that is not punctual and seemingly cannot be reproduced), or spatial ambiguity (a bug that cannot be localized to a component). We assume that Chopstix has been collecting data on one or more systems for which the problem is observed.

Chopstix data is searched to identify possible symptoms (unusual vital sign behavior). If the misbehavior has been reported to have occurred at specific times, then the visualizer can be used to zoom into the corresponding epochs at various time granularities. If the epoch of misbehavior does not yield an interesting symptom, then the information prior to these epochs is searched to find outliers in the vital signs. This is a fairly mechanical process and can be performed by setting threshold filters. Candidate symptoms are then correlated in the light of the observed misbehavior. If no diagnosis can be formulated as the symptoms do not bear any relationships, then the current set of symptoms is discarded and this step is repeated.

Once the problem has been diagnosed to a given set of symptoms, then it may either be reproduced by artificially triggering the symptoms, or alternatively avoided by working around the circumstances that led to the symptoms. The information available at this point should usually suffice to employ standard debugging tools.

3.2 Correlating Symptoms

Interpreting symptoms is the "art" of diagnosing system faults. Chopstix's value is in recording the vital signs, and visualizing those vitals to expose unusual behavior—i.e., symptoms. It is often through finding correlations across multiple symptoms that the root cause of the problem is understood.

While we cannot claim an automated "symptom correlator," we can formulate a set of rules to guide the diagnosis process. These rules typically depend on the components to which the symptoms in question apply, and thus require an understanding of the semantics of these components. Nevertheless, some rules are universal because they are inherent to the underlying OS. This section presents a collection of such rules, some of which have been implemented as filters and can be made to report these relationships in the visualizer.

Each rule maps one or more symptoms into the high-level behaviors that typically lead to such symptoms. Some rules are intuitive (e.g., a continuous increase in net memory allocated is a sign of a memory leak), while others are based on the repeated observation of a given cause and effect (e.g., a near-zero L2-cache-miss rate and heavy CPU utilization indicates a busy loop). The following classification of values as *low* and *high* is vital-sign-specific, and can be flagged by the visualization subsystem using appropriately set thresholds.

Rule #1: High CPU utilization ($cpuu$) with a low (nearly-zero) L2-cache-miss rate ($l2mi$) is a symptom of a busy loop; i.e., a loop that is unproductive but that does not yield the CPU to the scheduler. This is because processes in busy loops consume CPU resources but do not generate much memory traffic. The value of $l2mi$ gives a rough approximation of memory traffic, and when it is seen to be low with a high $cpuu$, it usually amounts to a busy loop.

Rule #2: An increase in the net (user or kernel) memory allocated is an indicator of a memory leak.

Rule #3: Unsatisfied I/O requests indicate bad blocks on the disk. An unsatisfied I/O request is defined as one for which the size of the data requested is greater than the size of the data returned in the corresponding response.

Rule #4: When the combined value of $cpuu$ for processes is low, scheduling delay ($sched$) is high, and the total $cpuu$ is high for the system, it is often a sign of a kernel bottleneck (e.g., a high-priority kernel thread). In this case, processes are not scheduled because the scheduler is not given a chance to run. This rule also applies to preemptable kernels, with the caveat that the symptoms are less conspicuous.

Rule #5: When the resource blocking vital ($blck$) for

a process is high, then it is typically mirrored by one or more of the following vitals: disk I/O ($blio$), locking ($lock$) or socket traffic ($sock$), thereby isolating the source of the blockage.

Rule #6: When the amount of user-memory allocated ($umem$) is high for a process, where $blck$ and $blio$ are also high for the same process, then it is a sign of low main memory and swap activity. When the latter two are low, it means that abundant main memory was available on the system at the time.

Rule #7: When $cpuu$ is low for a process and low for the kernel, and if $sched$ is high for the same process, then the process is being held from scheduling. If $cpuu$ is high for the system, then it is being held *normally*, in line with fair sharing of the CPU. If it is low, then it is being held *abnormally*.

Rule #8: When $kmem$ for a process is low compared to the rate of system calls ($syscall$), then it often indicates a busy loop with a system call; e.g., a `send` call that fails. The intuition is similar to that of Rule #1: A high value of $kmem$ is indicative of a process that causes heavy system activity. Accordingly, a high syscall rate should be mirrored by a high value of $kmem$.

Rule #9: Low $l2mi$ and high user-memory allocation ($umem$) in an application is a sign of application initialization. If $l2mi$ is high with a high $umem$ then it is a sign of increasing load. User memory is made available to processes on demand, when they access virtual-memory pages that have not yet been mapped to physical ones. At initialization time, processes "touch" several such pages while initializing data structures, but generate comparatively low memory traffic, leading to a low value of $l2mi$. Similarly, when memory is paged in and is accompanied by high memory traffic, then it is a sign of increasing load.

Rule #10: An abrupt disappearance of the vital signs of a process can indicate that it has died, or that it stopped functioning. The cause of death can be investigated by looking at the signals that it received and the syscalls it executed (e.g., in Linux, the `sys_exit` system call would indicate voluntary death, a `SIGSEGV` signal or a segmentation fault would indicate accidental death, and a `SIGKILL` signal would mean that the process was killed. In the case of a bug, the program counter associated with the stack trace can be used to trace the problem to the exact code site at which the problem was caused.

The remainder of this section uses these rules, along with other intuitions, to describe our experiences in debugging real problems on PlanetLab.

3.3 Case Study

This section presents a case study of an especially complex and troubling bug that we found using Chopstix. The PlanetLab team had been investigating it for over 6 months (Chopstix was not stable and had not been deployed up to this point) but was unable to pinpoint or reproduce the problem.

The example illustrates how the key features of Chopstix are leveraged: (1) multiple vital signs are correlated to understand the problem, (2) a long history saved through continuous data collection is mined for information, and (3) contextual information is used to rule out misdiagnoses and isolate anomalies to specific code sites.

Observed Behavior. Nodes were observed to crash every 1–7 days without leaving any information on the console or in the system logs. Shortly before such crashes it was observed that `ssh` sessions to nodes would stall for tens of seconds. Some nodes that were running an identical software stack did not suffer these crashes, indicating that the problem was load-dependent. KDB [24], a kernel debugger, was activated on one node in the hope that it would take over at the time of the lockup, but it did not survive the crash. The NMI watchdog was programmed to print a stack trace, but it printed program counters on the stack and often random false information.

Symptoms and Correlations (First Attempt). Visual inspection of the vital signs on nodes that had crashed showed spikes in the resource-blocking vital. Processes of the `ssh` daemon could be seen here, confirming that this accounted for the stalls on the `ssh` prompt. Rule #5 was applied and attributed the blocking to locking as well as I/O. To investigate the problem further, the stack traces for blocked processes on the I/O and lock signs were pulled out. The stack traces for locks resembled the following:

```
c0601655 in __mutex_lock_slowpath
c0601544 in mutex_lock
f885b1b4 in __log_wait_for_space
f8857e8e in start_this_handle
f8857f81 in journal_start
f889805b in ext3_journal_start_sb
f88938a4 in ext3_dirty_inode
c048a217 in __mark_inode_dirty
c04824f5 in touch_atime
```

Apparently, a process attempted to update the access time of a file, which led to an update of a journal transaction in the underlying file system. This code path can be deciphered to reveal that the journal transaction was protected by a locked mutex and was in the process of being committed to disk.

For I/O, the stack traces were similar to the one below:

```
c0600f1f in io_schedule
c048d922 in sync_buffer
c06010ff in __wait_on_bit
c0601181 in out_of_line_wait_on_bit
c048d887 in __wait_on_buffer
f885f908 in journal_commit_transaction
f8863128 in kjournald
c043167e in kthread
c0405a3b in kernel_thread_helper
```

This particular stack trace reveals the reason that the aforementioned batch of processes was blocked. The journal is trying to commit the current transaction to disk.

Many other processes were similarly blocked. It was speculated that the source of these blockages would also block all subsequent activities on the system and result in the crash. However, contrary to our expectations, it was found that in the epochs corresponding to this blockage, I/O throughput degraded negligibly and request-response latencies stayed low. Also, using a filter implementing Rule #3 to search this data for unsatisfied requests did not return any results adding to the evidence that the I/O subsystem was functioning normally. The clinching proof came with the observation that the activity was interrupted from time to time over many epochs during which there was little or no I/O, wherein many of the previously blocked processes would get unblocked and the blocking vital drop to a normal, low value.

Thus, the I/O "problems" were telltale and only a reflection of a legitimately heavy I/O workload. Chopstix was able to rule out the misdiagnosis through the correlation of multiple symptoms (I/O, blocking and locking) and through detailed hindsight that allowed the disk subsystem to be diagnosed postmortem.

Symptoms and Correlations (Second Attempt). Since the above symptoms did not yield a diagnosis, they were discarded. A new outlier was then discovered on the resource-blocking vital in the time shortly before the crashes. Some processes had been blocked for seconds, but this time Rule #5 did not apply. Furthermore, investigating stack traces showed that the blocking sites were not invocations of blocking operations. We wrote a filter to locate this effect throughout the available data set and found it interspersed throughout the history of the system. Closely coupled with these in the same epochs, were symptoms that matched Rule #4, that is, high scheduling delays with heavy CPU utilization in the kernel. On examining these spikes (the lead was now hinting at a bug in the scheduler) we found the corresponding bottleneck to be at line 6 of the following block of code.

```
1   try_unhold:
2     vx_try_unhold(rq, cpu);
3   pick_next:
4     if (unlikely(!rq->nr_running)) {
5       /* can we skip idle time? */
6       if (vx_try_skip(rq, cpu)) goto try_unhold;
7     }
```

A casual examination revealed this code to contain a tight loop. We then confirmed this problem by using kprobes to inject code at this site. This information was reported to the developers who wrote this code, which lies in the scheduler. Additional contextual information (e.g., the stack trace at this point) was also provided to them. They confirmed it to be a fault and went on to fix the bug.

In summary, the availability of a long history of multiple vital signs allows anomalies across them to be correlated, and access to contextual detail allows these anomalies to be traced to a specific code site. The application of these features to a problem that conventional approaches could not resolve is a validation of the diagnostic abilities of Chopstix.

3.4 Other Examples

We now briefly list other examples of diagnosis using Chopstix. Unlike the bug described in the case study, these problems were resolved relatively swiftly—usually within a day—as the relationships between the observed symptoms were more straightforward. To save space, we omit specific details and restrict ourselves to outlining the problem and the symptoms that led to the diagnosis.

SMP-dependent memory corruption in the kernel. By using a filter to find memory leaks, we were able to identify double deallocations of a socket structure provoked by an SMP race. This bug had prevented PlanetLab from using SMP hardware for over a year.

Hung file-system transactions. By correlating disk I/O, mutex locking and resource blocking, we were able to determine that a locking bug was poisoning the `inode` cache with uninitialized (locked) mutexes, causing filesystem misbehavior.

Out of low-memory crashes. Correlating spurts of I/O, kernel memory and blocking shortly before such crashes revealed that certain processes were unable to deallocate memory as they were blocked on I/O, causing memory to run out. The swap daemon was seen to be blocked as well, explaining why memory could not be reclaimed.

Watchdog restarts a failed program. By correlating disk I/O, syscalls, Socket activity and User-page allocation, we found that an application would fail to read its configuration file and crash, but that it would then be restarted immediately by a watchdog process implemented as part of the application. On each crash it would write out a log to disk, eventually filling the disk. Other methods (e.g., searching for large files on the disk) may have been used to identify this problem, but would have entailed a more rigorous examination of how the file, the config file and the watchdog interacted with one another than it was necessary with the global view provided by Chopstix.

`ps` *gets blocked accessing the kernel.* The blocking and I/O vital isolated unexplained blocking to a line in the kernel that was accessing the process command line. The explanation for the blocking was that the commandline is stored in the stack area of a process, which in this case was being swapped out for dormant processes.

4 Implementation

This section describes Chopstix's implementation. It first outlines the data structures, algorithms and design decisions involved in implementing the data collector. It then describes specific vital signs we have implemented using this data collector mechanism, and concludes with a brief overview of the aggregator and visualizer.

4.1 Data Collector

Sketch-based data collection involves five steps, the first four of which happen in the kernel: (1) a trigger fires for a potentially interesting event; (2) the relevant event data structure is retrieved and a sketch is updated; (3) a sampling function is evaluated to determine if this event should be recorded; (4) if so, a sample of relevant information is saved; and (5) a user-level process periodically polls the kernel for this data and resets the data structure, thereby defining a data collection *epoch*.

The sampling process is designed to balance two objectives. First, it should capture enough detail to identify interesting events. This is achieved by incorporating the notion of event-weights in the data structure that tracks per-event frequencies. A competing objective is to limit resource use to keep the overhead within predefined bounds. This is achieved through the use of an adaptive tuning mechanism that changes the effective sampling rate depending on the recent history of resource usage. The use of a logarithmic sampling function contributes to both objectives, minimizing the false negative rate to ensure accurate identification of important events, while reducing the resource consumption by aggressively reducing the sampling rate for events with large counts. We describe these features in detail below.

4.1.1 Event Triggers

The entry point into Chopstix is an *event trigger*, which invokes Chopstix in response to a change in a vital sign. One strategy to implement triggers is to poll various vital signs at set intervals and invoke Chopstix if they are seen to change significantly. In practice, however, it is always more efficient to instrument the OS interfaces that cause the relevant vital sign to change and invoke Chopstix on every invocation of such interfaces.

We illustrate this idea with an example. Kernel-page allocation may be monitored by polling the total number of free pages on the system. However, there may be periods during which there is no page allocation, making the polling process a waste of system resources. At other times, the page-allocation frequency may be higher than the corresponding polling rate, making the latter insufficient to account for each such event and leading to the limitations of uniform sampling. Thus, we instrument the interface to the page allocator to convey every page allocation to Chopstix as an event, which in turn accounts for each event in a sketch data structure and samples the events with better coverage than would be the case with the polling strategy. For monitoring hardware-related vital signs such as CPU utilization and L2-cache misses, the underlying processor can be made to generate interrupts to convey their values. The handlers for these interrupts are then used as placeholders for the event triggers corresponding to these vital signs.

4.1.2 Sketch Update

The sketch data structure employs a sampling function where the probability with which an event is sampled depends on its frequency. The frequency of events is approximated by counting the number of times the event occurs in a given data-collection epoch. Since maintaining exact per-event counts at the high frequency of vital signs is prohibitively expensive, we relax the requirement of exactitude in exchange for an approximate estimate that is relatively easy to obtain.

Chopstix uses a simple data structure—an array of counters indexed by a hash function (HCA)—to maintain these estimates, as well as to record samples collected for each vital sign. Updating an HCA is simple. Upon the reception of an event, the event label is extracted and hashed to generate an index for the HCA for the event type (vital sign). The label is specific to each vital sign, but typically consists of a virtual memory address, an executable identifier, and a user identifier. The counter at this index is incremented by the weight of the event (next paragraph). This counter is later used by the selection process to decide if a sample should be collected (next subsection).

An important observation about the events is that they are not all created equal. For example, iowait events can last from a few microseconds to several seconds. Clearly iowaits that run into seconds need to be assigned more "weight" in any performance analysis. Fortunately, the model of per-event counters can be extended easily to accommodate this requirement—while updating the HCA, the counters are incremented by a value that reflects the *weight* of the event being tracked. Returning to our I/O example, the length of the iowait period can be counted in jiffies (one jiffy is equal to the time period of the scheduler) and used as the weight associated with an iowait event. In this manner, call sites that are reached only a small number of times but account for a large amount of total time spent in iowait will get the same "count" as call sites that are reached more frequently but result in smaller aggregate iowait time.

Collisions in the hash function may cause two or more events to be hashed into the same location in the HCA.

We do not have any explicit mechanisms to handle collision as such a mechanism would impose an overhead that would be unsustainable at the targeted event frequencies. This decision makes the selection processes very fast and allows us to monitor a large number of high-frequency event-types with nearly zero overhead. It results in false negatives, but their probability is negligible, as documented in Section 5.

We considered other more precise data structures than HCAs, but were satisfied by the ensuing low probability of false negatives (i.e., important events that go unsampled) of HCAs and found their computational simplicity to be a good trade-off for the tighter bounds that come with more precise accounting data structures such as a Space-code Bloom Filters [16] and Counting Bloom Filters [7].

4.1.3 Sampling Function

Our objective in the selection process is to collect event samples for as many events as possible, without wasting resources on repetitive collection of the same (high-frequency) event. To achieve this, we use dynamic sampling probabilities computed as a function of the event-count in a given epoch. The function is chosen in such a way that the expected number of samples collected for any one event grows as $\log$ of the count of that event.

In practice, a simple threshold-based algorithm can achieve this effect. Using an integer sampling threshold t, the selection process selects an event if its estimated count is an integral power of t. The integer arithmetic can be simplified by picking a t that is a power of 2. The accuracy of the logarithmic sampling function is evaluated in Section 5.

To ensure bounded resource usage, Chopstix implements an adaptive scheme that tunes the threshold t. Chopstix uses a high watermark for the total resource consumption within an epoch, and doubles the value of t whenever the actual resource usage reaches this watermark. A low watermark is also used, with the value of t halved if resource usage at epoch boundaries is below this low watermark. The watermarks are defined in terms of the number of CPU cycles consumed by Chopstix, which are updated at the end of every epoch. These watermarks are configured by the administrator and can be used to set aside a fixed amount of CPU for monitoring.

4.1.4 Event Samples

When an event is selected for sampling, a variety of contextual information is collected and appended to a list of samples for the corresponding vital sign. The information includes a stack trace, the user identifier, a program identifier and other event-specific details that are descriptive of the circumstances at that point. The event-specific details usually include an operation field (e.g., allocation or deallocation for memory events, locking or unlocking for locks, read or write for I/O). Kernel stack traces are collected by following the current frame pointer.

User-level stack traces are collected similarly. However, since user memory may be paged out, and Chopstix being interrupt-safe is not allowed to block, it is first checked if the desired pages on the user stack are available in memory; if not, the trace is abandoned. In Linux, the executable file associated with a process is the first executable memory mapping in its address space and can be looked up quickly, but its path cannot be included in the sample as it can be arbitrarily long. To overcome this problem, we use a facility in Linux called `dcookie`, which stores the address of the data structure representing the file on disk (a `dentry` structure in Linux) in a hash table. We store the hash index in the sample and retrieve information about the data structure from user space at synchronization time at the end of every epoch.

4.1.5 Epochs

Data collection is carried out in fixed-sized epochs, under the control of a user-level process. The process invokes event-specific serialization functions on the sample list for each vital sign, and their output is saved, along with the HCAs from the previous epoch. The HCAs and all other data structures are then reinitialized. To avoid interruptions in monitoring at synchronization time, two copies of the HCA and sample arrays are maintained for each vital sign. At the end of each epoch, pointers are swapped and the unused HCAs are activated.

Having fixed epoch sizes allows us to circumvent an issue that arises with continuous monitoring. As data is accumulated while monitoring a running system, it also needs to be passed on for further processing. One approach is to identify certain entities, such as a process id, and export all data corresponding to an entity when the entity is no longer present in the system (i.e., export all events associated with a process when the process is terminated.). However, there are a few problems with this approach. First, data associated with entities is not available for analysis until the entity has expired. This may be acceptable during development but is a handicap when monitoring a production system. Second, the lifetime of entities is almost always some complex distribution that brings an additional layer of complexity if statistical analysis of the aggregate data is desired [6].

4.2 Vital Signs

The mechanisms just defined enable the efficient tracking of high-frequency system events. A set of vital signs

have been defined to use these mechanisms as a way to diagnose common problems on systems. An interface has been defined in the C language for allowing the extension of this set to define new event types. Since the processes downstream of event sampling do not depend on the number of event types monitored in the system, the extensions can be plugged in dynamically using instrumentation toolkits such as Dtrace [3] and Kprobes [20]. We now describe the implementation of the vital signs and the process of deploying new ones. Figure 2 tabulates the main characteristics of each vital sign, namely, where in the kernel it is implemented, the label used for hashing and the data collected when an event of this type is sampled.

CPU and L2-cache Utilization. CPU and L2-cache utilization are collected through hardware performance counter interrupts [25]. These counters are typically controlled through a set of model-specific registers (MSRs) which are used to select the type of event monitored and a counting threshold. When the number of processor events of a certain type exceeds this threshold value, a non-maskable interrupt (NMI) is generated and the value of the corresponding counter is conveyed to the OS. The Chopstix L2-cache and CPU triggers are located in the handler of this interrupt and trigger when an interrupt is delivered for the CPU-utilization and L2-cache-miss counters.

The event label used for the purpose of hashing consists of a PC, which is masked to drop the low 8 bits and an event subtype. The value of the PC can be obtained using MSRs and corresponds to the site whose CPU utilization or L2-cache misses is being measured. The PC is masked based on the observation that these events often register samples containing sites in close neighborhoods of each other and that are semantically dependent. Sketching on the other hand assumes tracked events to be independent, failing which it suffers from the weaknesses of uniform sampling [14]. By dropping the low bits of the PC, we reduce this dependency and make the collection of these events more robust. Note that the data collected when these events are sampled does not contain such an adjustment and instead uses the real value of the PC.

Scheduling Delay and Resource Blocking. Processes commonly get delayed in their normal course of execution by virtue of accessing resources that are not immediately available or as a result of scheduling policy. Rarely, as discussed in Section 3, processes may also get delayed due to system bugs. The scheduling delay and resource blocking vital signs correspond to the amount that various processes are delayed or blocked. The determination of whether a process is delayed or blocked is made based on the state of the process, which is changed by the kernel when a process accesses a resource that is unavailable. The trigger for these events lies in the routine that performs context switches between processes. On each context switch, the time since the last run of the process to be scheduled next is calculated. When this value exceeds a delay-qualifying threshold, a Chopstix event is generated. The time since the last run is calculated based on a timestamp, which is recorded by Chopstix in the context-switch routine at the time the process is scheduled, or when it undergoes a state transition.

The event label includes the site at which the process was preempted, which may be in the kernel. If in the kernel, computing this address entails tracing the stack and has an unacceptably high overhead for each context switch. Therefore, we use two-stage sampling, using only the pid in the first stage and tracing the stack to determine the value of the preemption site in the second stage.

Disk I/O and Page Allocation (Kernel and User). The Disk I/O trigger is located in the block I/O layer. On each such request, Chopstix determines the site of preemption and uses it as part of the event label. The data collected includes the site from which the I/O request was invoked. This site is not used in the event label as the number of such sites is small (10s to 100s) and the number that is actually used at a time is even smaller (5-10). The number of sites in the process at which disk I/O can be invoked is unlimited as a program and its data can be swapped out to disk and all system calls can lead to journal commits, which lead to disk activity. Therefore, using the latter in the event label yields a more uniform hash function.

Kernel page allocation uses the allocation site in the event label. The Chopstix trigger for this event is located in the lowest-level page-allocation interface so as to capture this event as exhaustively as possible. However, allocation sites in the kernel invoke these functions indirectly through higher-level interfaces which would normally make it necessary to trace the stack to determine the value of the call site. We cope with this situation by modifying the interface to these low-level page-allocation functions so that the calling site is passed to them as a parameter. The value of this parameter is used to construct a Chopstix event. The scope of these modifications was minor and involved less than 10 lines of changes to the relevant file.

User page allocation (or user memory allocation) is triggered in response to certain page faults. Such page faults are generated when a process touches memory that has been allocated to it, but that has not been mapped in. The context-switch site (i.e., the PC of the process at the

Vital sign	Trigger site	Event label	Data collected (please refer to the caption)
CPU utilization	NMI handler	Masked program counter (PC)	PC (not masked)
L2-cache misses	NMI handler	Masked PC	PC (not masked)
Scheduling delay	Context switch	Pid, site of pre-emption	Process priority and age
Resource blocking	Context switch	Pid, site of pre-emption	Process priority and age
Disk I/O activity	Block I/O requests	Pid, context-switch site,	Device attributes,request site, callback
Page allocation (kernel)	Page allocation and free routines	Pid,allocation or deallocation site	Amount of free memory (vector)
Page allocation (user)	Page fault handler	PC of page fault in process	Amount of free memory (vector)
System-call invocation	System-call handler	Masked stack pointer, syscall number	First argument and return value
Signal delivery	Signal-delivery function	Context-switch site,signal number	Address of handler
Socket data transfer	Filesystem read/write	Pid, call site	Destination address,port and protocol
Locking	Lock and unlock slowpath	Lock owner, locking or unlocking site	Number of waiters

Figure 2: Vital signs and their implementation. Note that the event label always includes an executable id and a user id, and that the data collected always includes the event label and a stack trace.

time it accessed the unavailable memory) is used in the event label here. Usually, such sites have been observed to lie in the neighborhood of calls to allocate memory in user-space (e.g., malloc). However, this may not necessarily be the case as there may be a significant lag between a process allocating memory and its actual use of it.

The data collected for both user and kernel allocations consists of the state of free memory for various regions, such as low memory, high memory and swap. This information can be used to study out-of-memory scenarios and at the same time reason about memory and disk traffic.

Syscalls, Signals and Socket Operations. Triggers to system calls and signals are implemented in the respective dispatchers. For system calls, the virtual memory address in the event label cannot correspond to the site from which the system call was invoked (i.e., the site of context switch) for the following reason. System calls are invoked via library functions that are typically implemented in a shared library. Thus, the calling site always has the same value, which corresponds to the `syscall` function defined in `libc` in Linux. We overcome this problem as follows. We would like to differentiate between system calls from different locations in the program. Since we cannot obtain this address, we will try to approximate it using a value that typically remains constant at a given code site. We have found the value of the stack pointer to work extremely well for this purpose. The reasoning behind this behavior is that the stack pointer reflects the execution path taken up to the execution of a system call, and that the number of unique paths that lead to system calls is relatively small for programs in general. Non-tail-recursive functions that execute system calls are rare but can unfortunately subvert this vital sign by flooding the sketch data structure, since each level of recursion is seen by Chopstix as a separate site. This case however, is easy to identify, even without inspecting the source code of the program in question, since the virtual memory addresses seen in the samples appear at constant offsets of each other.

The trigger for sockets sends and receives is a "catch-all" point for all types of sockets and lies in the filesystem layer. It is triggered when a file descriptor is read or written to, and when its file is of type socket. The type of data collected is the destination address, destination port and protocol number.

Locking. The locking vital involves mutexes and semaphores, which cause processes to sleep when they are accessed in locked state. Such processes are pushed into a queue and woken up sequentially when the owner of the lock unlocks the lock. Linux implements a fast-path for locks, which checks if the lock is available and acquires it by atomically testing its reference count and incrementing it. We do not modify this fastpath in the interest of its original design. Furthermore, the locking vital is interesting only when there is contention in the system, which is not the case when the fastpath succeeds. Accordingly, we modify the slow path of the locking and unlocking operations, which are invoked when an attempt is made to access a lock that is held by another process. We have modified the relevant lock data structures to contain a field identifying the owner of the lock, which is used in the event label as well as the data collected. When contention is noticed in the system, the owner can be referenced to determine the reason for the contention.

4.3 Data Aggregator

The goal of the aggregator is twofold. First, it organizes the information collected at multiple timescales, to filter them with a set of user-defined transformation functions and to pass the resulting information to the visualizer. Second, it *reconstructs* the sketch data structure for each epoch to compute the weighted sum of the vital signs for the epoch, and the total number of unique events seen (e.g., for memory allocation, these numbers correspond to the total memory allocated and the total number of memory-allocation sites). These actions happens

in two stages: a processing phase in which the data is aggregated and transformed once and for all and cached, and a retrieval phase in which contents of the cache are returned to the requesting user. Queries for arbitrary intervals are supported but not cached. Cached retrievals are faster than un-cached ones by about a factor of 10.

The aggregator is invoked by the visualizer when a user requests the visualization of a given time range of activity. The main data structure used by the aggregator is a list of queues. The number of entries in the list corresponds to the number of timescales (currently three). As mentioned previously, the data collected by Chopstix is organized as a set of timestamped directories. The data aggregator reads data from these directories in chronological order. As data is read in, it is also aggregated and transformed. Transformation functions are implemented as queue operators, are compiled into shared libraries and are scanned in when the aggregator is called into operation by the visualizer. When a transformation function is added, changed or removed, all caches are invalidated.

4.4 Visualizer

The visualizer is web-based and is implemented in Macromedia Flash. The aggregator returns the information requested by the visualizer in two parts. The magnitudes of vital signs for each epoch are URL-encoded and the output of the transformation functions consisting of samples including program counters, stack traces and other contextual information are returned in XML format. The magnitudes of the vital signs are plotted in line graphs in separate panes that can be overlapped. When a given epoch is clicked on, the corresponding samples for the active timescale are displayed.

5 Evaluation

The effectiveness of Chopstix relies on its low overhead, which enables it to run continuously; and its extensive coverage of the events monitored, which prevents potentially interesting behaviors from flying under the radar. This section evaluates these two key properties, along with the responsiveness of the Aggregator and the Visualizer. We measure the data collection overhead through micro- and macro-benchmarks, comparing the macro-benchmarking results to those obtained for a comparatively information-poor system profiler. We quantify coverage in terms of the probability of false negatives for which we derive a formula. We verify empirically that this probability is low in practice.

For our experiments, we use a system with a Core2Duo processor and 4GB of RAM running Linux version 2.6.20.1. The performance counter hardware was configured to generate NMI interrupts for every 10^7 CPU cycles and for every $6 * 10^5$ L2-cache misses. The configuration of Chopstix was set to disable all limits on the CPU utilization that are otherwise used in adaptive sampling.

Test	Chopstix disabled	Chopstix enabled	Slowdown
getpid	0.1230	0.1263	2.6%
read	0.2427	0.2422	0.2%
write	0.2053	0.2054	<0.1%
stat	0.9738	0.9741	<0.1%
fstat	0.3175	0.3195	0.6%
open/close	1.64	1.66	1.2%
select (10 fds)	0.7411	0.7498	1.2%
select (500 fds)	15.3139	15.3649	0.3%
signal handler	1.565	1.540	<0.1%
protection fault	0.432	0.432	<0.1%
pipe latency	4.3461	4.3940	1.1%
fork+exit	69.185	66.8312	<0.1%
fork+execve	259.336	259.61	<0.1%

Table 1: LMbench: with and without Chopstix

5.1 Data Collection

Table 1 reports the results of running the lmbench [18] micro-benchmark with and without Chopstix. The main takeaway is that the overhead is negligible. It exceeds 1% only for simple operations for which the sketch update is not amortized, but these operations are being executed in a busy-loop that is uncommon and usually considered to be an anomaly in a production environment.

Table 2 displays the results of two macro-benchmarks: a kernel compile and an HTTP server. These tests compare the performance of a Chopstix-monitored system with that of a vanilla system, as well as with the performance of a system running OProfile [12]. Oprofile was configured to track only two events—CPU utilization and L2-cache utilization—both at the same rate as Chopstix. It was configured to collect stack traces in its profiles. Chopstix was configured to collect all the vitals and auxiliary information described throughout this paper.

The kernel compile was carried out using the following steps: clean the kernel tree (`make mrproper`), configure the kernel (`make oldconfig`), reboot the system, and compile it. For the web-server benchmark, we used a tool named `apachebench`, which sends requests to the benchmarked system over a fixed number of connections specified at the command line. The number of connections was set to 512, and a single HTML file of about 1400 bytes was used as a target. We ran both the client and the server on the same machine to eliminate the dependence of the experiment on the network. Both tests were conducted 10 times for each configuration.

As shown in Table 2, Chopstix has a near-zero overhead. Moreover, when compared to Oprofile, it is effec-

Test	Chopstix disabled	Chopstix enabled	OProfile
Kernel compile	213 secs ($\pm 4secs$)	214 secs ($\pm 4secs$)	370 secs ($\pm 13secs$)
apachebench	4080req/sec ($\pm 75/sec$)	4080req/sec ($\pm 75/sec$)	3229req/sec ($\pm 75req/sec$)

Table 2: Kernel compile and web-server performance.

tively an order of magnitude more efficient on the kernel-compile benchmark and significantly more efficient on the HTTP benchmark. This, in spite of the fact that Chopstix collects significantly more information. Some of the microbenchmark results give an edge to the performance of the system with Chopstix enabled. We verified that this difference was the result of two different kernels being used, and that disabling Chopstix did not influence it.

5.2 Aggregation/Visualization

The responsiveness of the combined aggregator and visualizer was measured by requesting a data set spanning three days via the visualizer, first in uncached mode and then in cached mode. This process was repeated 10 times. The aggregator was configured to use six filters to post-process the data.

The initialization of the session to the aggregator, during which the requested data is read and post-processed, took about 80 seconds when it was not cached and about 3 seconds when it was cached. In both cases, once the session had been initialized and the data was available in the memory of the aggregator, queries for contextual information about specific epochs completed instantaneously.

5.3 Coverage of Sketches

We now analyze the probability that a qualifying event is missed by our method of sampling. Consider an event with frequency n, where the corresponding counter reaches a value of N. Note that N is never smaller than n. The counter value N could exceed n if one or more *other events* with total frequency $N - n$ collided with it. The probability that $N > n$ is dependent of the actual distribution of the event frequencies. While such a distribution is not known in general, we can use the total number of events (say M) observed during a monitoring interval, and the size of the HCA (number of counters, say m) to estimate the probability of collisions. Given such a distribution, the probability of false negatives can be shown to be: $P[unsampled|N-n,n] = \binom{N-n}{k}\left(\frac{N-n}{N}\right)^k$ where k is $\lfloor \log_2 N - t \rfloor$, and t represents the threshold below which nothing is sampled. Thus, the probability of false negatives can be manipulated by varying the value of t and the size of the HCA. In this way, these values can be adjusted so that qualifying events are sampled with a very high probability even when they collide with noise events in the HCA.

Next, we experimentally evaluate this probability in the context of a PlanetLab dataset by maintaining exact state for a small set of events, for which collisions are resolved. The probability of false negatives for a given vital sign is calculated as an average over these events. At the time of initial deployment of Chopstix, trial and error with this experiment was used to fix the value of the HCA data structure for various vital signs. In our current configuration, in which HCAs vary between 128 bytes and 4 kilobytes in size, the probability of false negatives for each of the vital signs lies between 10^{-3} and 10^{-4}. That is, Chopstix captures over 99.9% of the relevant events.

6 Discussion

For a problem to be diagnosed using Chopstix, it needs to satisfy two properties. First, it must have a measurable effect on the system's behavior. A logical bug that causes the generation of incorrect outputs in a program falls out of scope, *unless* the incorrect outputs are associated with a deviation in behavior; e.g., higher latency, smaller size of data leading to lower memory traffic, and so on. Second, the system must stay up for at least one Chopstix epoch from the time of the first manifestation of the problem. "Red-button conditions" that happen instantly whenever the appropriate trigger is set off, are not covered by Chopstix. Sometimes, even with these properties satisfied, the problem may not be captured on the vital signs in sufficient detail to yield a diagnosis. Still, our experience is that situations in which no progress is made at all in the diagnosis are rare.

Chopstix requires the kernel running on monitored nodes to be extended with a patch that implements the event triggers. The data collection mechanism is implemented as a pluggable kernel module. CPU utilization and L2-cache-miss performance counters should be supported by the processor for the corresponding vital signs to be activated. Software into which stack-trace-visibility is desired needs to be compiled with frame-pointer support. In order for instruction addresses to be translated into code sites, programs need to include debugging information—and their source code needs to be available on the centralize analysis server where the aggregator runs—not on the monitored node.

We do not believe that Chopstix needs any special OS support that ties it to Linux (or UNIX), as the vital signs of Chopstix are common across all OSes. Still, there are specific features, such as the retrieval of a pathname using a directory cookie, which will likely have to be reimplemented if it is ported to another OS. Event labels and event samples may also have to be populated other-

wise depending on the specific implementation of callback functions and request-response semantics in the target OS. However, the high-level information conveyed by these items should stay the same.

7 Related Work

A plethora of diagnostic tools are available. They are related to Chopstix through the mechanism they implement (e.g., profilers, monitors) or through their common purpose (e.g., interactive debuggers, deterministic replay tools, dynamic instrumentation tools). The former are geared more towards performance tuning than failure diagnostics and have fundamental differences at the mechanistic level, in spite of similarities. For example, Chopstix uses sketches to streamline its data and is hence able to collect numerous *types* of data in extensive detail. To the best of our knowledge, Chopstix is the first diagnostic tool to use such data collection technique. As for the latter set, Chopstix differentiates itself by attacking problems that cannot be reproduced or localized. To this end, Chopstix can be used in conjunction with many of these tools by helping to localize and characterize problems that can then be debugged using standard tools.

More specifically, DCPI [1] is a *profiler* designed to run continuously on production systems. The goal of DCPI is *profiling* to optimize system performance, not *diagnosis* to find faults and anomalies. This objective makes it focus on processor events that aid in uncovering inefficiencies at the instruction level, answering questions such as "how long did a particular instruction stall on average because of a cache miss". Its post-processing and visualization tools are also tailored to this purpose. Samples are accounted for deterministically and restricted to the process id and program counter. OProfile [12] is an implementation of the DCPI concept for Linux. We evaluate the performance of Chopstix against that of OProfile in Section 5.

Ganglia [8] is a cluster-monitoring tool that collects system-wide totals of several high-level variables; e.g., CPU utilization, throughput, free disk space for each monitored node. CoMon [21] is a distributed monitoring tool deployed on PlanetLab that is similar to Ganglia in its functioning. At the usability level, the difference between Chopstix and these tools is in the depth of information available and the type of variables monitored. Chopstix focuses on low-level variables and saves contextual detail while CoMon and Ganglia focus on high-level variables (uptime, boot state, free disk etc.) and track only system-wide totals. At the functional level, they serve a different purpose. The role of tools like CoMon and Ganglia is to help flag misbehaviors (e.g., "a node went down") while the role of Chopstix is to explain such misbehaviors.

Dtrace [3], kprobes [20] and JIT-i [19] facilitate the insertion of custom code into the OS kernel at run time. Such code can be used to perform arbitrary actions, but it is typically used to collect debugging information for fault diagnosis. These tools are difficult to apply to the type of problems addressed by Chopstix as they require the search space first be narrowed to a set of candidate code sites to be instrumented. Furthermore, there is no explicit provision to manage large amounts of data if the instrumented site is executed with a high frequency. On the other hand, these tools function well as companions to Chopstix, as they can be used to instrument code sites that Chopstix identifies as being of interest.

Recently, techniques have been developed to deal with system bugs without requiring an intervention from a programmer. Triage [27] and Rx [23] are tools that use lightweight re-execution support to respond to bugs in this way. Triage revisits the point of failure repeatedly in an attempt to isolate the condition that led to the bug. Rx on the other hand tries to eliminate such conditions. Both of these tools require a characterization of a problem to the extent of attributing it to a given application and defining the condition of failure (e.g., a segmentation fault at a given virtual address). This point reaffirms the value of Chopstix in providing such characterizations and completing the chain of diagnosis.

There have also been several tools for deterministic replay [9, 13, 26]. These are alternative approaches to using Chopstix. These tools are typically invasive, imposing high CPU overheads, generating large volumes of data and requiring that systems be shut down for diagnosis. They are thus less appropriate for use on production systems. Other tools collect execution histories in the same way Chopstix does [28, 17]. However, the type of information they collect and the types of symptoms they handle are quite different from the ones in Chopstix.

Recent research has proposed the use of statistical clustering techniques to classify system state as being normal or erroneous [4, 5]. There are two points to make about this work. First, as currently envisioned, these techniques depend on user-defined *Service-Level Objective* functions to indicate when the state of the system is inconsistent with respect to low-level properties such as CPU utilization. This method is limited to a set of problems for which Service-Level Objectives are defined and have been annotated with a root-cause. In contrast, Chopstix enables diagnosis based on analysis of comprehensive event logs, enabling root-cause analysis in cases for which a objective functions do not (yet) exist. Second, looking beyond current techniques, we believe that Chopstix can leverage machine learning to enhance problem detection by

automating the correlation between system failures and the collected vital signs. Such techniques might also be used to automate the correlation of symptoms that reveal root causes.

8 Conclusion

This paper describes Chopstix, a tool that helps track down the intermittent, non-reproducible problems that often make maintaining production systems a challenge. Chopstix captures misbehaviors by logging all behavior on the system in the form of succinct summaries that contain logs of low-level OS events. We think of these events as representing the system's vital signs, and the diagnosis problem as one of looking for symptoms that correspond to unusual variations in (and correlations among) these vitals. Using Chopstix, it is possible to isolate intermittent failures in enough detail to allow traditional debugging tools to be applied.

The key enabler for Chopstix is a randomized data structures called sketches, which results in a negligible overhead (1% CPU utilization) and high coverage (99.9% of the relevant events). We have used Chopstix to troubleshoot problems that we were not able to diagnose using existing tools.

The next step for Chopstix is to look for correlations across multiple monitored nodes to find network-wide bugs. We would also like to correlate Chopstix data with network-level flow logs, and to allow applications to be modified to add to the contextual state of event samples collected. Finally, we would like to add more automation to the post-processing tool-chain, perhaps through the use of machine learning techniques to classify symptoms.

Acknowledgements

We gratefully acknowledge feedback from the reviewers, our shepherd, Anthony Joseph, Andy Bavier, Charles Consel, Julia Lawall, Murtaza Motiwala, Jennifer Rexford and Vytautas Valancius. This work was funded, in part, by NSF grants CNS-0335214 and CNS-0520053.

References

[1] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, 1997.

[2] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, June 1990.

[3] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, 2004.

[4] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase. Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 16–29, 2004.

[5] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 105–118, 2005.

[6] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a better netflow. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 245–256, 2004.

[7] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. In *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, 1998.

[8] Ganglia Development Team. Ganglia monitoring system. URL: http://ganglia.info.

[9] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, 2006.

[10] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[11] Intel. VTune performance analyzer homepage: developer.intel.com/software/products/vtune/index.html.

[12] John Levon et al. OProfile - a system profiler for ulinux. URL: http://oprofile.sourceforge.net/doc/index.html.

[13] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.

[14] A. Kumar, M. Sung, J. Xu, and J. Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution, 2002.

[15] A. Kumar and J. Xu. Sketch guided sampling - using on-line estimates of flow size for adaptive data collection. In *Proc. IEEE INFOCOM*, 2006.

[16] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li. Space-code Bloom filter for efficient per-flow traffic measurement. In *Proc. IEEE INFOCOM*, Mar. 2004.

[17] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2003.

[18] L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 1996.

[19] M. Olszewski, K. Mierle, A. Czajkowski, and A. D. Brown. JIT instrumentation: a novel approach to dynamically instrument operating systems. *SIGOPS Oper. Syst. Rev.*, 41(3), 2007.

[20] P. S. Panchamukhi. Kernel debugging with kprobes: Insert printk's into the linux kernel on the fly. URL: http://www.ibm.com/developerworks/library/l-kprobes.html.

[21] K. Park and V. S. Pai. Comon: a mostly-scalable monitoring system for planetlab. *SIGOPS Oper. Syst. Rev.*, 40(1):65–74, 2006.

[22] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences building planetlab. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, 2006.

[23] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies—a safe method to survive software failures. *ACM Trans. Comput. Syst.*, 25(3), 2007.

[24] SGI. KDB - built-in kernel debugger. URL: http://oss.sgi.com/projects/kdb.

[25] B. Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):64–71, 2002.

[26] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2004.

[27] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: diagnosing production run failures at the user's site. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007.

[28] C. Verbowski, E. Kiciman, A. Kumar, B. Daniels, S. Lu, J. Lee, Y.-M. Wang, and R. Roussev. Flight data recorder: monitoring persistent-state interactions to improve systems management. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, 2006.

[29] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, 2006.

[30] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. System support for automatic profiling and optimization. *SIGOPS Oper. Syst. Rev.*, 31(5):15–26, 1997.

Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions

Xu Chen[§] Ming Zhang[†] Z. Morley Mao[§] Paramvir Bahl[†]
[†] *Microsoft Research* [§] *University of Michigan*

Abstract – Large enterprise networks consist of thousands of services and applications. The performance and reliability of any particular application may depend on multiple services, spanning many hosts and network components. While the knowledge of such dependencies is invaluable for ensuring the stability and efficiency of these applications, thus far the only proven way to discover these complex dependencies is by exploiting human expert knowledge, which does not scale with the number of applications in large enterprises.

Recently, researchers have proposed automated discovery of dependencies from network traffic [8, 18]. In this paper, we present a comprehensive study of the performance and limitations of this class of dependency discovery techniques (including our own prior work), by comparing with the ground truth of five dominant Microsoft applications. We introduce a new system, Orion, that discovers dependencies using packet headers and timing information in network traffic based on a novel insight of delay spike based analysis. Orion improves the state of the art significantly, but some shortcomings still remain. To take the next step forward, Orion incorporates external tests to reduce errors to a manageable level. Our results show Orion provides a solid foundation for combining automated discovery with simple testing to obtain accurate and validated dependencies.

1 Introduction

Modern enterprise IT infrastructures comprise of large numbers of network services and user applications. Typical applications, such as web, email, instant messaging, file sharing, and audio/video conferencing, operate on a distributed set of clients and servers. They also rely on many supporting services, such as Active Directory (AD), Domain Name System (DNS), Kerberos, and Windows Internet Name Service (WINS). The complexity quickly adds up as different applications and services must interact with each other in order to function properly. For instance, a simple webpage fetch request issued by a user can involve calls to multiple services mentioned above. Problems at any of these services may lead to failure of the request, leaving the user frustrated and IT managers perplexed.

We say one service *depends* on the other if the former requires the latter to operate properly. Knowledge of service dependencies provides a basis for serving critical network management tasks, including fault localization, reconfiguration planning, and anomaly detection. For instance, Sherlock encapsulates the services and network components that applications depend on in an *inference graph* [8]. This graph is combined with end-user observations of application performance to localize faults in an enterprise network. When IT managers need to upgrade, reorganize, or consolidate their existing applications, they can leverage the knowledge of dependencies of their applications to identify the services and hosts that may potentially be affected, and to prevent unexpected consequences [9]. When continually discovered and updated, dependencies can help draw attention to unanticipated changes that warrant investigation.

While there are network management systems that perform topology and service discovery [21, 12], IT managers currently do not have proven tools that help to discover the web of dependencies among different services and applications. They commonly rely on the knowledge from application designers and owners to specify these dependencies. These specifications can be written in languages provided by commercial products, such as Mercury MAM [4] and Microsoft MOM [5]. While straightforward, this approach requires significant human effort to keep up with the evolution of the applications and their deployment environment. This becomes a massive problem for large enterprises with thousands of applications. For example, a survey conducted by the Wall Street Journal in 2008 found that HP and Citigroup each operate over 6,000 and 10,000 line-of-business (LOB) applications [6]. Microsoft runs over 3,100 LOB applications in its corporate network, most

of which have no documentations describing their dependencies. More recently, there have been a few attempts to automate dependency discovery by observing network traffic patterns [8, 9, 18]. However, there is very little understanding about *how well these approaches work, where and how they fall short, and whether their limitations can be overcome without human intervention.*

There are a few challenges in designing a system that discovers dependencies in a complex enterprise network: First, it should require minimal human effort; Second, it should be applicable to a diverse set of applications; Third, it should be non-intrusive to applications and be easily deployable; and Fourth, it should scale with the number of services, applications, and hosts in the network. These challenges are hard to address, especially given that expert knowledge of application internals cannot be assumed for thousands of new and legacy applications. Incorporating such knowledge in a system is a formidable task.

We have built a system called Orion that overcomes all these challenges. Specifically, it discovers dependencies by passively observing application traffic. It uses readily available information contained in IP, TCP, and UDP headers without parsing higher-level application-specific protocols. Most of the computation is done locally by individual hosts, and the amount of information exchanged between hosts is small.

In this paper, we describe our experiences in designing, implementing, and deploying Orion in Microsoft's corporate network. We make the following contributions:

- We introduce a new dependency discovery technique based on traffic delay distributions. For the applications we studied, we can narrow down the set of potential dependencies by a factor of 50 to 40,000 with negligible false negatives.
- We are the first to extract the dependencies for five dominant enterprise applications by deploying Orion in a portion of Microsoft's corporate network that covers more than 2,000 hosts. These extracted dependencies can be used as input to create more realistic scenarios for the evaluation of various fault localization and impact analysis schemes
- We comprehensively study the performance and limitations of a class of dependency discovery techniques that are based on traffic patterns (including Orion). The results reveal insights into the shortcomings of such techniques when they are applied to real-world applications.
- We conduct extensive experiments to compare Orion with the state of the art (Sherlock [8] and eXpose [18]). While their false negatives are similar, the false positives of Orion are 10-95% fewer than Sherlock and 94-99% fewer than eXpose. Even though Orion cannot avoid all the false positives, we can obtain accurate dependencies using simple external tests.

In the rest of the paper, we elaborate on our techniques, implementation, and evaluation of automated dependency discovery. Additionally, we provide concrete examples about how to use extracted dependencies for fault diagnosis and reconfiguration planning.

2 Related Work

Many sophisticated commercial products, such as EMC SMARTS [1], HP OpenView [2], IBM Tivoli [3], Microsoft MOM [5], and Mercury MAM [4], are used for managing enterprise networks. Some of them provide support for application designers to specify the dependency models. However, these approaches require too much manual effort and are often restricted to a particular set of applications from the same vendor.

There is a large body of prior work on tracing execution paths among different components in distributed applications. For example, Pinpoint instruments the J2EE middleware on every host to track requests as they flow through the system [15]. It focuses on mining the collections of these paths to locate faults and understand system changes. X-Trace is a cross-layer, cross-application framework for tracing the network operations resulting from a particular task [16]. The data generated by X-Trace can also be used for fault detection and diagnosis. Both Pinpoint and X-Trace require all the distributed applications to run on a common instrumented platform. This is unlikely to happen in large enterprise networks with a plethora of applications and operating systems from different vendors.

Magpie is a toolchain that correlates events generated by operating system, middleware, and application to extract individual requests and their resource usage [10]. However, it heavily relies on expert knowledge about the systems and applications to construct schemas for event correlation.

Project5 [7] and WAP5 [22] apply two different correlation algorithms to message traces recorded at each host to identify the causality paths in distributed systems. They both focus on debugging and profiling individual applications by determining the causality between messages. The message correlation in Project5 is done by computing the cross correlation between two message streams. WAP5 developed a different message correlation algorithm based on the assumption that causal delays follow an exponential distribution for wide-area network applications. In contrast, Orion focuses on discovering the service dependencies of network applications.

Brown *et al.* propose to use active perturbation to infer dependencies between system components in distributed

applications [14]. While this methodology requires little knowledge about the implementation details of the applications, it has to use a *priori* information to learn the list of candidate services to perturb, which is inherently difficult to obtain in large enterprise networks.

The closest prior work to Orion is the use of traffic co-occurrence to identify dependencies in enterprise networks. To determine whether one service depends on the other, researchers have tried to compute either the conditional probability [8, 9] or the JMeasure [18] of the two services within a fixed time window. A key issue with both approaches is the choice of the time window size. In fact, it is fundamentally difficult to pick an appropriate window size that attains a good balance between false positives and false negatives. While they seem to extract *certain* meaningful dependencies, neither of them quantified the accuracy of their results in terms of how many true dependencies they missed or how many false dependencies they mistakenly inferred. In contrast, our technique does not rely on any co-occurrence window size. Through field deployment, we show that Orion extracts dependencies much more accurately than Sherlock [8] and eXpose [18] for a variety of real-world enterprise applications. We also validated our results with the owners of all these applications.

3 Goal & Approach

Given an enterprise network application, our goal is to discover the set of services on which it depends in order to perform its regular functions. Before describing the technical details, we first introduce a few concepts and terms that will be used in the paper. We then motivate our design decisions, outline our approach, and discuss our challenges.

3.1 Services and dependencies

Enterprise networks consist of numerous services and user applications. Applications, such as web, email, and file sharing, are directly accessed by users. Most applications depend on various network services to function properly. Typical network services include Active Directory (AD), Domain Name System (DNS), and Kerberos. These services provide basic functions, such as name lookup, authentication, and security isolation. An application or a service can run on one or more hosts.

In this paper, we do not make a formal distinction between services and applications, and we use both terms interchangeably. We use a three-tuple $(ip, port, proto)$ to denote either an application or a service. In an enterprise network, an ip normally maps to a unique host and the $port$ and $proto$ often identify a particular service running on that host. Many ports under 1024 are reserved for well-known services, such as Web (80, TCP), DNS (53, TCP/UDP), Kerberos (88, TCP/UDP), WINS (137, TCP/UDP), and LDAP (389, TCP/UDP). Another type of service is RPC-based and does not use well-known ports. Instead, these services register an RPC port between 1025 and 65535 when a host boots up. Clients who intend to use these services will learn the RPC service port through a well-known port of RPC endpoint mapper (135).

While it is a common practice to associate a service with an $(ip, port, proto)$ tuple, we may define service at either coarser or finer granularities. On the one hand, many enterprises include fail-over or load balancing clusters of hosts for particular services, which can be denoted as $(ipCluster, port, proto)$. Other services, such as audio and video streaming, could use any port within a particular range, which can be denoted as $(ip, portRange, proto)$. On the other hand, multiple services may share the same port on a host in which case we must use additional service-specific information to identify each of them.

We define service A to depend on service B, denoted as $A \rightarrow B$, if A requires B to satisfy *certain* requests from its clients. For instance, a web service depends on DNS service because web clients need to lookup the IP address of the web server to access a webpage. Similarly, a web service may also depend on database services to retrieve contents requested by its clients. Note that $A \rightarrow B$ does not mean A must depend on B to answer *all* the client requests. In the example above, clients may bypass the DNS service if they have cached the web server IP address. The web server may also bypass the database service if it already has the contents requested by the clients.

3.2 Discovering dependencies from traffic

We consider three options in designing Orion to discover dependencies of enterprise applications: i) instrumenting applications or middlewares; ii) mining application configuration files; and iii) analyzing application traffic. We bypass the first option because we want Orion to be easily deployable. Requiring changes to existing applications or middlewares will deter adoption.

Configuration files on hosts are useful sources for discovering dependencies. For instance, DNS configuration files reveal information about the IP addresses of the DNS servers, and proxy configuration files may contain the IP addresses and port numbers of HTTP and FTP proxies. However, the configuration files of different applications may be stored in different locations and have different formats. We need application-specific knowledge to parse and extract dependencies from them. Moreover, they are less useful in identifying dependencies that are dynamically constructed. A notable example is that web browsers often use automatic proxy discovery protocols to determine their proxy settings.

In Orion, we take the third approach of discovering dependencies by using packet headers (*e.g.,* IP, UDP, and

TCP) and timing information in network traffic. Such information is both easy to obtain and common to most enterprise applications. Note that it is natural to develop application-specific parsers to understand the application traffic, *e.g.,* when a message starts or ends and what the purpose of the message is. Such detailed knowledge is helpful in determining the dependency relationships between the traffic of different services, eliminating ambiguities, and hence improving the accuracy of dependency inference. Nonetheless, developing parsers for every application requires extensive human effort and domain knowledge. For this reason, we refrain from using any packet content information besides IP, UDP, and TCP headers.

Orion discovers dependencies based on the observation that *the traffic delay distribution between dependent services often exhibits "typical" spikes that reflect the underlying delay for using or providing these services.* While conceptually simple, we must overcome three key challenges. First, it is inherently difficult to infer dependencies from application traffic without understanding application-specific semantics. Packet headers and timing information are often insufficient to resolve ambiguity. This may cause us to mistakenly discover certain *service correlations* (false positives) even though there are no real dependencies between the services. Second, packet headers and timing information can be distorted by various sources of noise. Timing information is known to be susceptible to variations in server load or network congestion. Third, large enterprise networks often consist of tens of thousands of hosts and services. This imposes stringent demand on the performance and scalability of Orion. We introduce new techniques to address each of the three challenges.

Orion has three components. The *flow generator* converts raw network traffic traces into flows. The purpose is to infer the boundaries of application messages based only on packet headers and timing information. The *delay distribution calculator* identifies the potential services from the flows and computes delay distributions between flows of different services. Finally, the *dependency extractor* filters noise and discovers dependencies based on the delay distributions. We describe each of them in detail in the subsequent sections.

4 Flow Generation

In client-server applications, services and their clients communicate with each other using requests and replies. For convenience, we use a *message* to denote either a request or a reply. Orion discovers service dependencies by looking for the time correlation of messages between different services. For instance, it infers the dependency of a web service on a DNS service by observing DNS messages precede web messages. While time correlation may not always indicate a true dependency, we rely on a large number of statistical samples to reduce the likelihood of false positives.

In reality, we are only able to observe individual packets in the network instead of individual messages. Multiple packets may belong to the same message and the time correlation among themselves do not explicitly convey any dependency information. If we consider the time correlation between every possible pair of packets, we could introduce: i) too much redundancy because we count the correlation between two dependent messages multiple times; and ii) significant computation overhead because the number of packets is much larger than the number of messages.

While it is desirable to aggregate packets into messages for dependency inference, this is nontrivial because we do not parse the application payload in packets. Given that most services use UDP and TCP for communications, we aim to both reduce computation overhead and keep sufficient correlation information by aggregating packets into *flows* based on IP, TCP, and UDP headers and timing information:

TCP packets with the same five tuple (locIP, locPt, remIP, remPt, proto) are aggregated into a flow whose boundary is determined by either a timeout threshold, or TCP SYN/FIN/RST, or KEEPALIVE. Any two consecutive packets in a flow must not be interleaved by an interval longer than the timeout threshold. TCP SYN/FIN/RST flags are explicit indications of the start or the end of flows. Certain services with frequent communications may establish persistent connections to avoid the cost of repetitive TCP handshakes. They may use KEEPALIVE messages to maintain their connections during idle periods. We also use such messages to identify flow boundaries.

UDP packets with the same five tuple (locIP, locPt, remIP, remPt, proto) are aggregated into a flow solely based on timing information, since UDP is a connectionless protocol. Any two consecutive packets in a flow must not be interleaved by an interval longer than the timeout threshold.

We will evaluate the impact of flow generation on our inference results in Section 7.2.3.

5 Service Dependency Discovery

In this section, we first present an overview of our approach to discovering service dependencies. We then describe the details of our approach, including how to calculate delay distributions between different services based on flow information and how to extract dependencies from delay distributions.

5.1 Overview

Orion discovers service dependencies by observing the time correlation of messages between different services.

Our key assumption is if service A depends on service B, the delay distribution between their messages should not be random. In fact, it should reflect the underlying processing and network delays that are determined by factors like computation complexity, execution speed, amount of communication information, and network available bandwidth and latency. For instance, a web client may need to go through DNS lookup and authentication before accessing a web service. The *message delay* between the DNS and web services is the sum of: 1) the time it takes for the client to send an authentication request after the DNS reply is received; 2) the transmission time of the authentication request to the authentication service; 3) the processing time of the authentication request by the authentication service; 4) the transmission time of the authentication reply to the client; and 5) the time it takes for the client to send a web request after the authentication reply is received. Assuming the host and network load are relatively stable and relatively uniform service processing overhead, this message delay should be close to a "typical" value that exhibits as a "typical" spike in its delay distribution.

There could be multiple typical values for the message delay between two dependent services, each of which corresponds to a distinct execution path in the services. In the above example, the client may bypass the authentication if it has a valid authentication ticket cached. As a result, the message delay between the DNS and web services will simply be the time it takes for the client to send a web request after the DNS reply is received. This will lead to two typical spikes in the delay distribution.

While there could be thousands of hosts in the network, Orion focuses on discovering service dependencies from an individual host's perspective. Given a host, it aims to identify dependencies only between services that are either used or provided by that host. This implies the dependency discovery algorithm can run independently on each host. This is critical for Orion to scale with the network size. By combining the dependencies extracted from multiple hosts, Orion can construct the dependency graphs of multi-tier applications. The dependency graphs of a few three-tier applications are illustrated in Section 7.1.

In the remainder of the section, we will describe a few important techniques in realizing Orion. This includes how to scale with the number of services, reduce the impact of noise, and deal with insufficient number of delay samples.

5.2 Delay distribution calculation

Orion uses the delay distribution of service pairs to determine their dependency relationship. Given a host, we use $(IP_{loc}, Port_{loc}, proto)$ and $(IP_{rem}, Port_{rem}, proto)$ to represent the local and remote services with respect to that host. We are interested in two types of dependency: i) Remote-Remote (RR) dependency indicates the host depends on one remote service to use another remote service. This type of dependency is commonly seen on clients, *e.g.,* a client depends on a DNS service $(DNS_{rem}, 53_{rem}, UDP)$ to use a web service $(Web_{rem}, 80_{rem}, TCP)$; ii) Local-Remote (LR) dependency indicates the host depends on a remote service to provide a local service. This type of dependency is commonly seen on servers, *e.g.,* the web service on a server $(Web_{loc}, 80_{loc}, TCP)$ depends on an SQL database service $(SQL_{rem}, 1433_{rem}, TCP)$ to satisfy the web requests from its clients.

Orion calculates the delay distribution based on the flow information generated in the previous stage (Section 4). Since a host may observe many flows over a long time, Orion uses two heuristics to reduce the CPU and memory usage. First, it calculates the delays only between flows that are interleaved by less than a predefined time window. Ideally, the time window should be larger than the end-to-end response time of any service S in the network (from the time a client sends the first request to a service that S depends on till the time the client receives the first reply from S) to capture all the possible dependencies of S. In single-site enterprise networks, a time window of a few seconds should be large enough to capture most of the dependencies that we look for, given the end-to-end response time of services in such networks is typically small. In multi-site enterprise networks which are interconnected via wide-area networks, we may need a time window of a few tens of seconds. We currently use a three-second time window for our deployment inside Microsoft's corporate network.

The second heuristic to reduce overhead is based on the observation that a host may communicate over a large number of services, many of which may not be persistent enough to attract our interest. For instance, clients often use many ephemeral ports to communicate with servers and the "services" corresponding to these ephemeral ports are never used by other services. Orion keeps track of the number of flows of each service in the recent past and uses a flow count threshold to distinguish between ephemeral and persistent services. It calculates and maintains delay distributions only for persistent services pairs. The flow count threshold is determined by the minimum number of statistical samples that are required to reliably extract dependencies. We use a default threshold of 50 in the current system. Note that the window size and the flow count threshold only affect the the computation and storage overhead but not the accuracy of Orion.

Since Orion does not parse packet payload to understand the actual relationship between flows, it simply calculates the delay between every pair of flows, *e.g.,* $(LocIP_1, LocPt_1, RemIP_1, RemPt_1, proto_1)$ and

$(LocIP_2, LocPt_2, RemIP_2, RemPt_2, proto_2)$, that are interleaved by less than the time window. We treat each delay sample as a possible indication of both a RR dependency, *e.g.,* $(RemIP_2, RemPt_2, proto_2) \rightarrow (RemIP_1, RemPt_1, proto_1)$, and an LR dependency, *e.g.,* $(LocIP_1, LocPt_1, proto_1) \rightarrow (RemIP_2, RemPt_2, proto_2)$, and add it to the delay distributions of both service pairs. This implies that there could be "irrelevant" samples in the delay distribution that do not reflect a true dependency between the service pair. This could be problematic if a delay distribution is dominated by such irrelevant samples. Nonetheless, in our current deployment, we identified only one false negative that is possibly caused by this problem (Section 7.2).

Suppose a host uses m remote services and provides n local services, Orion needs to maintain delay distributions for $(m \times m)$ RR service pairs and $(m \times n)$ LR service pairs for that host in the worse case. Because Orion discovers dependencies for each host independently, m and n are determined by the services observed at that host rather than all the services in the network. This allows Orion to scale in large enterprises with many services. We evaluate the scalability of Orion in Section 7.4.1.

5.3 Service dependency extraction

We now describe three important issues related to extracting dependencies from delay distributions: mitigating the impact of random noise, detecting typical spikes, and dealing with insufficient samples.

5.3.1 Noise filtering & spike detection

The delay distribution of service pairs calculated from flow information is stored as a histogram with a default bin width of 10ms. There are 300 bins if we use a three-second time window. We denote *bin-height* as the number of delay samples that fall into each bin.

Raw delay distributions may contain much random noise due to host and network load variations. The noise will introduce numerous random spikes in the delay distribution, which could potentially interfere with the detection of typical spikes. Realizing this problem, we treat each delay distribution as a signal and use signal processing techniques to reduce random noise. Intuitively, the number of typical spikes corresponds to the number of commonly-executed paths in the services, which is at most a few for all the services we study. In contrast, random noise tends to introduce numerous random spikes in the signal, which is more evident in the high frequency spectrum.

This prompts us to use Fast Fourier Transform (FFT) to decompose the signal across the frequency spectrum and apply a low-pass filter to mitigate the impact of random noise [13]. The choice of low-pass filter reflects the trade-off between tolerance to noise and sensitivity

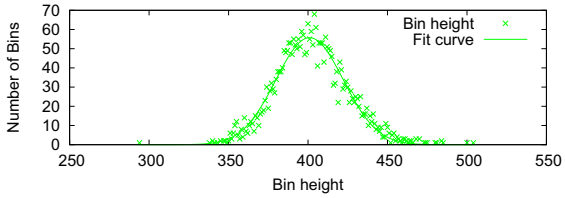


Figure 1: *Bin-heights fit well with normal distribution*

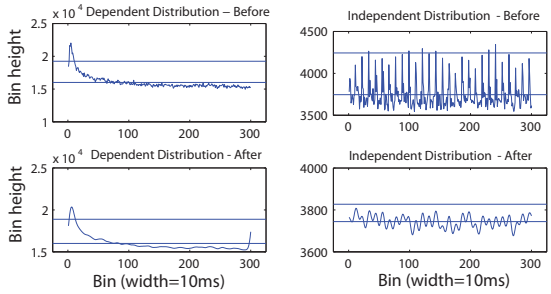


Figure 2: *Delay distributions before and after filtering*

to typical spikes. We have tried a few commonly-used filters and find that Kaiser window ($50 \leq \beta \leq 200$) [17] achieves a reasonable balance between the two goals. The effect of filtering is not particularly sensitive to the choice of β within the above range and we use $\beta = 100$ in the current system.

For each delay distribution, we plot the corresponding bin-height distribution. Each point in the bin-height distribution represents the number of bins with a particular bin-height. Interestingly, we find these bin-height distributions closely follow normal distribution, as illustrated by an example in Figure 1. Based on this observation, we detect typical spikes whose bin-heights are among the top $x\%$ in the bin-height distribution. The parameter x determines the degree of tolerance to noise and sensitivity to typical spikes. In practice, we find x between 0.1% and 1% works pretty well. We use a bin-height threshold of $(mean + k \times stdev)$ to detect typical spikes, where $mean$ and $stdev$ are the the mean and standard deviation of the bin-heights. With $k = 3$, we will detect typical spikes whose bin-heights are among the top 0.3% in the bin-height distribution.

Figure 2 shows two examples of noise filtering and spike detection. The two horizontal lines in each graph represent the $mean$ and the $(mean + k \times stdev)$ of the bin-heights. The two graphs on the left are the delay distributions of a true dependency before and after filtering. Clearly, filtering does not eliminate the typical spike. The two graphs on the right are the delay distributions of a non-dependency. In this case, filtering significantly reduces the random spikes that could have led to false positives. Note that noise filtering is effective only against random spikes in delay distributions. It has little effect on other non-typical spikes introduced by certain unexpected service pair interaction.

5.3.2 Client & service aggregation

Orion requires a reasonably large number of samples in a delay distribution to reliably detect typical spikes. To avoid inaccuracy due to a lack of samples, it ignores delay distributions in which the number of samples is fewer than the number of bins in the histogram. This could be problematic for individual clients who use many remote services infrequently. Fortunately, clients in an enterprise network often have similar host, software, and network configurations. They also have a similar set of dependencies when using a particular remote service. Orion aggregates the delay distributions of the same service pairs from multiple clients to improve the accuracy of dependency extraction. Note that the service dependencies of clients may have slight difference, *e.g.,* due to different software versions. By doing client aggregation, Orion will discover the aggregated dependencies of all the clients, which could be a superset of the dependencies of each individual client.

To facilitate client aggregation, we may have to perform service aggregation as well. Many enterprise networks use a failover or load balancing cluster of servers to provide a particular service. Clients may communicate with any of the servers in the cluster. By treating such a cluster of servers as a whole and representing the service with a $(ipCluster, port, proto)$, it provides us much more opportunities in performing client aggregation. Similarly, a server may provide the same service (*e.g.,* audio and video streaming) on any port in a particular range. We may represent such a service with $(ip, portRange, proto)$ to help client aggregation.

While client and service aggregations help to improve accuracy, they require extra information beyond that embedded in the packet headers. In Microsoft's corporate network, most servers are named based on a well-defined convention, *e.g.,* xxx-prxy-xx is a proxy cluster and xxx-dns-xx is a DNS cluster. We develop a simple set of naming rules to identify the clusters. We also examine the configuration files to obtain the port range for a few services that do not use a fixed port. In enterprises where such naming convention does not exist, we may have to rely on IT managers to populate the host-to-cluster mapping information. Normally, this type of information already exists in large enterprises to facilitate host management. We can also leverage existing work on service discovery to obtain this information [11].

5.4 Discussion

We focus on discovering the service dependencies for client-server applications, which are dominant in many enterprise networks. Their dependencies change only when they are reconfigured or upgraded. As a result, the dependencies that we aim to discover are usually stable over several weeks or even months. As we will see in Section 7.3, this is critical because Orion may need a few days of statistical samples to reliably infer dependencies. Recently, peer-to-peer (p2p) applications have gained popularity in enterprise networks. In contrast to traditional client-server applications, they are designed to be highly resilient by dynamically changing the set of hosts with which a client communicates. The dependencies of these applications could change even during short periods of time. As future work, we plan to investigate how to discover the dependencies of p2p applications.

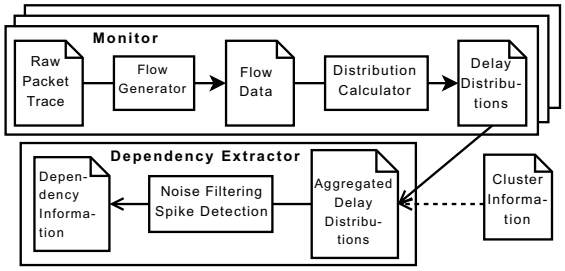


Figure 3: *System architecture*

Orion discovers service dependencies by looking for typical spikes in the delay distributions of service pairs. While conceptually simple, false positives and false negatives may arise due to various types of noise (*e.g.,* different hardware, software, configuration, and workload on the hosts and load variation in the network) or unexpected service pair interaction (*e.g.,* although service $A \rightarrow B$, the messages of A and B could be triggered by other services). While the impact of random noise can be mitigated by taking a large number of statistical samples, unexpected service pair interaction is more problematic. In Section 7.2, we will illustrate examples of false positives where non-dependent service pairs show strong time correlations.

We emphasize that the issues above are not just specific to Orion but to the class of dependency discovery techniques based on traffic patterns. We will comprehensively evaluate and compare their performance using five real-world enterprise applications in Section 7. In spite of these issues, Orion is surprisingly effective in discovering service dependencies. In fact, it not only discovers the majority of the true dependencies but also successfully eliminates most of the false positives. While some false positives are unavoidable, their numbers are sufficiently small to be removed with some simple testing.

Orion requires a large number of statistical samples to reliably extract service dependencies. This makes it less applicable to services which are newly deployed or infrequently used. It may also miss dependencies that rarely occur, such as DHCP. One possible solution is to proactively inject workloads to these services to help accumulate sufficient number of samples.

6 Implementation

We now describe the implementation of Orion as shown in Figure 3. Orion has three major components that run on a distributed set of hosts in an enterprise network. The *flow generators* convert raw traffic traces into flow records in real time. The *delay distribution calculators* run on the same set of hosts as the flow generators. They continually update the delay distributions for all the service pairs relevant to the services that administrators are interested in. A centralized *dependency extractor* collects and analyzes the delay distributions from multiple hosts to extract dependencies.

In a fully distributed deployment, each host runs a flow generator and a delay distribution calculator to build its own delay distributions (host-based deployment). Such organization scales well given the localized nature of computation. Traffic traces can be captured by WinPcap or TDI drivers (a Windows API). The latter allows us to get the port and protocol information even when traffic is encrypted by IPSec. When such a fully distributed deployment is not possible, Orion can operate on packet sniffers connected to span ports on switches and routers that are close to hosts (network-based deployment). It will build the delay distributions for each host on the same subnet.

6.1 Flow generator

A flow generator reads the $(ip, port, proto)$ and timing information from the raw traffic traces and outputs flow records. It maintains a hash table in memory, which keeps track of all the active flow records using the five-tuple $(locIP, locPt, remIP, remPt, proto)$ as keys. $locIP$ corresponds to a monitored host. Each flow record contains a small amount of information, *e.g.,* the timestamps of the first and the last packets, the direction and TCP flag of the last packet, and the current TCP state of the flow. Based on this information, we can determine whether to merge a new packet into an existing flow record, flush an existing flow record, or create a new one. To keep the hash table from growing excessively, we expire old flow records periodically. The current version is implemented in C using the libpcap library with roughly 2K lines of code.

6.2 Delay distribution calculator

The delay distribution calculator keeps a buffer that holds the recent flow records of each monitored host. The flow records in the buffer are sorted based on their starting time and ending time. When a new flow record arrives, we use its starting time and ending time minus the three-second time window to expire old records in the buffer. For each monitored host, we also maintain a set of delay distributions for the service pairs related to that host. We go through all the existing flow records in the buffer, compute the delay between the new flow record and each of the existing ones, and insert the delay samples into the delay distributions of the corresponding service pairs. Each delay distribution is maintained as a histogram with 300 bins and 10ms bin width. We implement this component using Perl with roughly 500 lines of code.

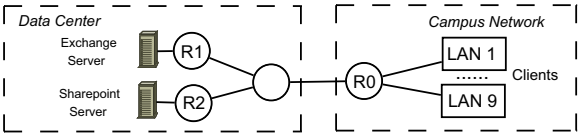


Figure 4: *Deployment in Microsoft corporate network*

6.3 Dependency extractor

The centralized dependency extractor waits for dependency extraction requests for a particular service from administrators. When a request arrives, it will retrieve the delay distributions of relevant service pairs from the servers where the service is hosted and the clients. Depending on whether there are enough samples to reliably extract dependencies, the dependency extractor may perform client and service aggregation when clustering information is available. Aggregation is done by adding the bin-heights of the same bins in the delay distributions of the same service pair. After retrieving and possibly aggregating the delay distributions, we ignore those delay distributions with fewer samples than the number of bins. For each remaining delay distribution, we use Matlab to perform Fast Fourier Transform, filter noise with Kaiser window, and then detect typical spikes whose bin-heights exceed $(mean + k \times stdev)$. If any typical spike exists, we consider the corresponding service pair a dependency and output the list of all the dependencies in the end. We use a combination of Perl and Matlab codes for aggregation, noise filtering, spike detection, and report generation, with a total of 1K lines of code.

7 Experimental Results

We deployed Orion in a portion of Microsoft's corporate network illustrated in Figure 4. Because we cannot directly access the clients and the production servers, we choose an network-based deployment by connecting packet sniffers to span ports on routers. We monitored the traffic of 2,048 clients in 9 LANs at router R_0 and the traffic of 2 servers in the data center at routers R_1 and R_2. The client traffic must traverse R_0 to reach the data center, where most of the services are hosted. From the traffic at R_0, we extract the RR dependencies for five representative applications from the client's perspective. By examining the traffic at R_1 and R_2, we extract the LR dependencies for two of the five applications from the server's perspective. The results in this section were obtained during a two-week period in January 2008. We thoroughly evaluate Orion in its accuracy of dependency extraction, its convergence properties, and its scalability and performance.

Type	Sharepoint	DFS	OC	SD	Exchg
# of Instances	1693	1125	3	34	34
# of Clients	786	746	228	196	349

Table 1: Popularity of five enterprise applications

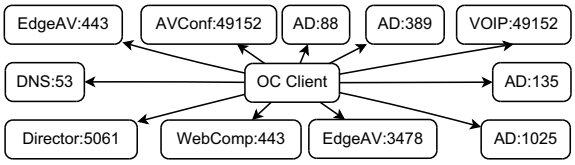


Figure 5: *OC client dependencies*

7.1 Dependencies of five applications

Microsoft's corporate network has thousands of applications. We select five distinct applications based on their popularity. These five applications include Office Communications (integrated instant messaging, VoIP, and audio and video conferencing), Exchange (email), Sharepoint (web), Distributed File System (file serving), and Source Depot (version control system).

For each application, Table 1 lists the number of clients and the number of application instances of the same type based on the traffic at R_0. Clearly, each application attracts a reasonably large fraction of the monitored clients. There are also many application instances of the same types in the network. Since the same type of applications have similar dependencies, our results may be easily extended to many other application instances. For each application, we obtain its true dependencies from the deployment documents written by the application owners. This is one key distinction from previous work which does not have access to such ground truths to perform comprehensive validations. Note that due to the large amount of time and effort involved, application owners can only create these documents for a small subset of important applications.

There are four infrastructural services that most applications depend on. Among them, active directory (AD) and proxy services are provided by load balancing clusters and DNS and WINS services are provided by failover clusters. We aggregate all the servers in the same cluster and represent each service as an $(ipCluster, port, proto)$. Since most services support both UDP and TCP, we omit the $proto$ field for simplicity in the remaining of this section. We next describe the service dependencies of the five applications studied based on the ground truths from deployment documents also confirmed by their application owners.

7.1.1 Office communications (OC)

Office Communications (OC) is an enterprise application that combines instant messaging, VoIP, and audio and video (AV) conferencing. It is one of the most popular applications and is being used by 50K+ users in Microsoft's corporate network. Figure 5 illustrates the dependencies of OC clients. They depend on eleven services to exploit the full functionality of OC: 1) DNS:53 for server name lookup during login; 2) Director:5061 for load-balancing login requests; 3) AD:88 for user authorization; 4) AD:389 for querying relevant domain objects; 5) AD:1025 (an RPC port) for looking up user profile; 6) AD:135 for learning the port number of the AD:1025 service; 7) EdgeAV:3478 for AV conferencing with external users via UDP; 8) EdgeAV:443 for AV conferencing with external users via TCP; 9) AVConf:49152 for AV conferencing with internal users; 10) VoIP:49152 for voice-over-IP; 11) WebComp:443 for retrieving web contents via HTTPS.

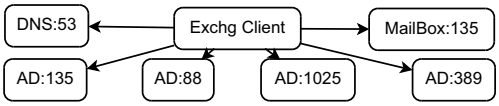


Figure 6: *Exchange client dependencies*

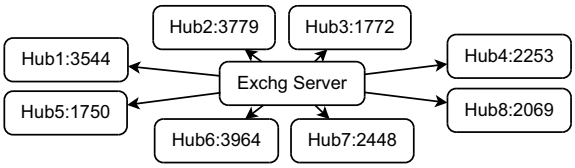


Figure 7: *Exchange server dependencies*

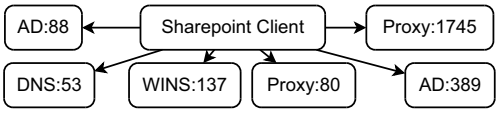


Figure 8: *Sharepoint client dependencies*

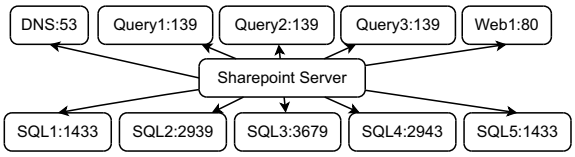


Figure 9: *Sharepoint server dependencies*

7.1.2 Exchange

Exchange is an enterprise email application. It is being used by all the users in Microsoft. Figure 6 and 7 illustrate its client and mailbox server dependencies. Exchange clients depend on six services to use the email service, five of which have been explained before. Because clients use RPC to communicate with the email service, it also depends on the endpoint mapper service on the mailbox server (Mailbox:135) to learn the RPC port of the email service. The email service on the mailbox server depends on eight services to answer the requests from Exchange clients, each of which is an email submission service running on a hub transport server. Note that we can obtain a three-tier dependency graph of Exchange by combining the client-side dependencies with the server-side dependencies.

7.1.3 Sharepoint

Sharepoint is a web-based enterprise collaboration application. We studied one of the most popular internal

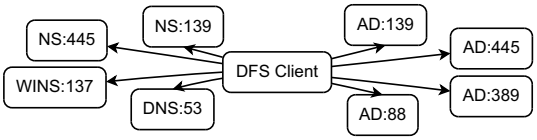


Figure 10: *DFS client dependencies*

Figure 11: *SD client dependencies*

Sharepoint websites. Figures 8 and 9 illustrate its client and front-end server dependencies. Sharepoint clients depend on six services to use the web service, three of which have been explained before. The remaining three services are: 1) WINS:137 is required because the web service uses a NetBios name which can only be looked up via WINS; 2) Proxy:80 is for notifying clients with proxy settings that Sharepoint is an internal website; 3) Proxy:1745 is for notifying clients without proxy settings that Sharepoint is an internal website. The web service on the front-end server depends on ten services to answer requests from clients: 1) five SQL services that store most of the web contents; 2) one web service that stores the remaining web contents; and 3) three query services that handle search requests. We can obtain a three-tier dependency graph of Sharepoint by combining the client-side dependencies with the server-side dependencies.

7.1.4 Distributed file system (DFS)

DFS is an enterprise service that can organize many SMB file servers into a single distributed file system. We study one of the DFS services where internal users can download most of the installation packages of Microsoft softwares. Figure 10 illustrates the dependencies of DFS clients. They depend on eight services to access the files in DFS, four of which are unique to DFS. AD:445 and AD:139 help clients find the DFS namespace servers (NS). NS:445 and NS:139 redirect clients to the appropriate file servers.

7.1.5 Source depot (SD)

Source depot (SD) is a CVS-like version control system. We study one of the SD services that is frequently used by our monitored clients. Figure 11 illustrates the service dependencies of SD clients. There are only four dependencies, all of which have been explained before.

7.2 Accuracy of dependency discovery

We first examine the accuracy of the dependencies discovered by Orion for each of the five applications depicted above. We then further remove false positives with additional testing, compare our results with prior work based on co-occurrence probability, and study the effects of noise filtering and flow generation.

For each application, we first create a *candidate set* of services that the application could possibly depend on if we do not make any inference. For a client-side or a server-side application, it is simply the full set of the remote services that the client or the server ever communicates with. We classify the services in the candidate set into true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) by comparing the inferred dependencies with the true dependencies presented in the previous section.

The rows starting with Orion in Table 2 and 3 present the breakdown of the client-side (with aggregation) and server-side dependencies respectively. The number of services in the candidate sets varies from several hundreds to hundreds of thousands for different applications, reflecting the difficulty in extracting the dependencies manually. Orion can narrow down the set of potential dependencies ($TP + FP$) to fewer than a hundred, making it much easier to identify the true dependencies with some additional testing. This represents a factor of 50 to 44K reduction from the original candidate sets. Furthermore, we only miss two true dependencies on the server side (Table 3), one for each application. There is no false negative for any of the applications on the client side (Table 2).

For the server-side Sharepoint web service, we miss one dependency on a database service. Further investigation indicates that there is no typical spike in the delay distribution between the two services, likely due to the noise induced by the background indexing traffic between the two services which are unrelated to the web service. For the server-side Exchange email service, we miss the dependency on one of the eight email submission services. While visual inspection does reveal a typical spike in the corresponding delay distribution, it is not significant enough to be caught by our spike detection.

The number of FP's varies from 3 for the client-side Sharepoint service to 77 for the client-side OC service. They fall into two categories depending on whether they contain any significant, non-typical spikes in their delay distributions. The FP's in the first category are unlikely to be caused by random noise. Manual inspection indicates most of these non-typical spikes can be explained by the existence of certain *correlation* between the service pairs. As one example, the OC client has false dependency on the exchange email service, apparently due to many clients running both applications simultaneously. In another example, the Exchange client has false dependency on the proxy service. This can happen when Exchange clients open emails with embedded external web contents, causing these clients to access external websites via proxy. The FP's in the second category are apparently due to the limitation of our spike detection algorithm to fully distinguish noise from spikes.

	Exchange client				DFS client				Sharepoint client				OC client				SD client			
	tp	fp	fn	tn	tp	fp	fn	tn	tp	fp	fn	tn	tp	fp	fn	tn	tp	fp	fn	tn
Orion	6	26	0	14K	8	13	0	1497	6	3	0	703	11	77	0	25K	4	4	0	369
Sher_{10}	6	178	0	14K	8	102	0	1408	6	65	0	641	9	125	2	25K	4	52	0	321
Sher_{100}	6	57	0	14K	8	93	0	1417	6	168	0	538	10	85	1	25K	4	29	0	344
eXpose	5	443	1	14K	8	570	0	940	6	565	0	141	10	1416	1	24K	4	323	0	50
noFilter	6	49	0	14K	8	25	0	1485	6	6	0	700	11	159	0	25K	3	19	1	354
noFlow	6	2488	0	12K	8	988	0	522	6	534	0	172	11	3594	0	21K	4	198	0	175

Table 2: Client side dependencies after aggregation

	Exchange server				Sharepoint server			
	tp	fp	fn	tn	tp	fp	fn	tn
Orion	7	34	1	230K	9	6	1	660K
Sher_{10}	8	68	0	230K	8	7	2	660K
Sher_{100}	7	61	1	230K	9	19	1	660K
eXpose	7	557	1	230K	7	396	3	660K
noFilter	4	44	4	230K	6	3	4	660K

Table 3: Server side dependencies

7.2.1 Removing false positives

In the process of extracting dependencies from the candidate set, we have to trade off between FP's and FN's. Our primary goal is to avoid FN's even at the expense of increasing FP's. This is because we have almost no way to recover a true dependency once it is removed from the candidate set. In cases where dependencies are used for fault localization or reconfiguration planning, missing dependencies may lead to unanticipated consequences that are expensive to diagnose and repair. (see Section 8 for details).

To further reduce the FP's in Table 2, we perform controlled experiments on the client side. For each of the five applications, we use a firewall to block the services in the FP and TP sets one-by-one. Blocking the services in the FP set will not have any impact on the application while blocking the services in the TP set will disrupt its service function. To eliminate caching effect, we must start with a clean state for each test. Because this is a manual process, it took us roughly one working day to successfully identify all the 35 true dependencies from the 158 potential ones. We did not conduct such experiments on the server side because we have no control over the servers. Administrators can do such testing during maintenance hours to minimize the disruption to users. Note that developing test cases requires human knowledge of only how to drive applications but not of application internals. The former is relatively widely available while the latter is usually arduous to extract.

7.2.2 Comparison with Sherlock & eXpose

Sherlock [8] and eXpose[18] attempt to extract dependencies from network traffic. They are both based on the idea that the traffic of dependent services are likely to co-occur in time. They use a fixed time window W to compute co-occurrences and then use a threshold T either on the conditional probability (in Sherlock) or on the JMeasure (in eXpose) to identify dependencies. While they both seem to extract certain meaningful dependencies, neither of them quantified the accuracy of their results in terms of false negatives or false positives.

While conceptually simple, a key problem with both approaches is the choice of W. As we explained earlier in Section 7.2.3, the delay between two dependent services reflects the underlying processing and network delay. This delay could vary from a few milliseconds to hundreds of milliseconds. If W is small (as in Sherlock), we may miss the dependencies between the service pairs whose typical delays exceed W. If W is large (as in eXpose), we are likely to capture many co-occurrences of non-dependent service pairs. In contrast, Orion identifies dependencies by looking for typical spikes in the delay distributions. It does not make any assumption about the location of the typical spikes in the distribution.

We implemented both Sherlock and eXpose for comparison. We use $W = 10ms$ and $100ms$ for Sherlock and the $W = 1s$ for eXpose. (In their papers, Sherlock uses $W = 10ms$ and eXpose uses $W = 1s$.) We tune their threshold T so that their FN's roughly match ours, and then compare the FP's. The results are in the rows starting with Sher_{10}, Sher_{100}, and eXpose in Tables 2 and 3. Clearly, Orion has far fewer FP's than Sherlock or eXpose in all the cases. For the client side of Exchange, DFS, Sharepoint, and SD, the FP's inferred by Orion are only 5% - 50% of those inferred by Sherlock or eXpose. Assuming that the testing time to remove FP's grows linearly with the number of potential dependencies, Orion will save approximately two to twenty days of human testing time compared with Sherlock and eXpose.

7.2.3 Effect of noise filtering & flow generation

Orion relies on noise filtering to mitigate the impact of random noise on dependency discovery. It is important to understand to what extent noise filtering helps to reduce FP's and FN's. In Table 2 and 3, the results in the rows starting with "noFilter" are computed by applying spike detection directly to the delay distributions without filtering noise. Judging from the Orion results, noise filtering is effective in reducing FP's and/or FN's in all but one case. Even in the Sharepoint server case where Orion has 3 more FP's, we consider it worthwhile given the decrease of 3 FN's.

Orion aggregates packets into flows to reduce redun-

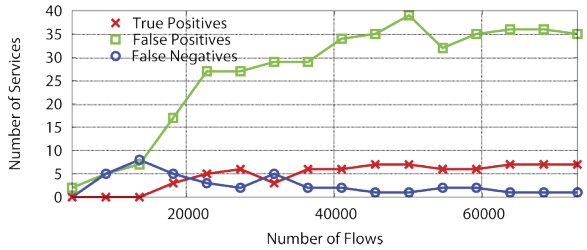


Figure 12: *Impact of flows on Exchange server dependencies*

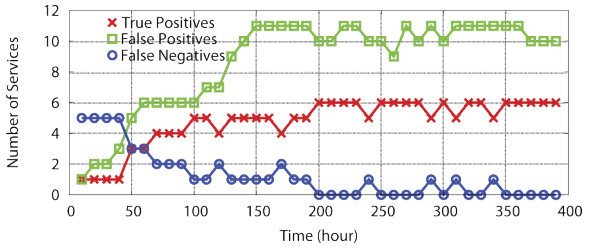


Figure 13: *Impact of time on Exchange client dependencies*

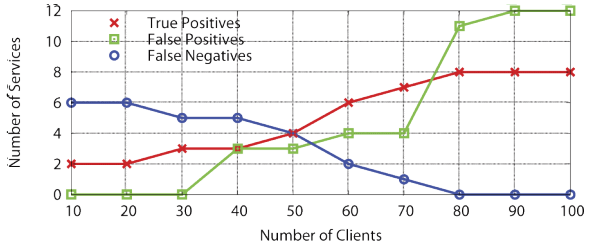


Figure 14: *Impact of aggregation on DFS client dependencies*

dant correlation and computation overhead. Without flow generation, big flows are treated more favorably than small flows since they will contribute more samples in the delay distributions. Such systematic bias may lead to undesirable spikes in the delay distribution. In Table 2, the row starting with "noFlow" contains the results without flow generation. Compared with the Orion results, FN's stay the same but FP's are much larger, most likely due to the over-counting of delay samples related to big flows. In terms of performance, calculating delay distribution directly from packets is roughly ten times slower than from flows. This is because there are significantly more packet pairs than flow pairs.

7.3 Convergence of dependency discovery

We now study the convergence properties of Orion along three dimensions: time, flows, and clients. This is important because the dependencies of an application may change from time to time due to reconfiguration. We evaluate whether Orion can produce stable results before the next change happens. Furthermore, Orion discovers dependencies based on the delay samples computed from flows. Understanding its requirement on the number of flows is essential for us to judge the reliability of the results. Finally, Orion sometimes needs client aggregation to overcome the problem of a lack of sufficient samples. Measuring the impact of clients helps to avoid unnecessary overhead due to excessive client aggregation.

Figure 12 illustrates how the inferred dependencies of Exchange server change as more flows to the Exchange server are used for dependency discovery. The X-axis is the number of flows. The number of samples in all the delay distributions related to the Exchange service grows with the number of flows. Clearly, Orion can discover more TP's when more flows are used. When the number of flows reaches 65K, Orion discovers all the TP's and the number of FP's also stabilizes. This suggests that the variation of inferred dependencies ($TP + FP$) is a good indication of whether Orion needs more flows. The convergence behavior of other applications exhibits similar trend. Depending on the application, Orion needs 10K to 300K flows to obtain stable results.

Figure 13 shows how the inferred dependencies of Exchange client evolve over time. Not surprisingly, the accuracy of the inferred dependencies gradually improves as Orion uses longer duration of traces. After 200 hours, it has discovered all the TP's and the inferred dependencies fluctuate only slightly thereafter. This confirms that we can stop the inference when the inferred dependencies converge. For all the applications, the convergence time varies from two to nine days. We consider this acceptable since the dependencies of production client-server applications are usually stable for at least several weeks to several months in enterprise networks. Nonetheless, this convergence time could be a bit long for newly deployed applications. We may expedite the discovery process by applying dependency templates derived from other networks or provided by application designers to pre-filter the set of possible dependencies.

Figure 14 illustrates how the inferred dependencies of DFS client vary as we aggregate more clients. It is evident that client aggregation is important for improving the inference accuracy, especially when no individual clients have a sufficient number of delay samples. The FN's drop from 5 to 0 as the aggregated clients increase from 10 to 90. After that, the inferred dependencies become stable even when more clients are aggregated. This suggests excessive client aggregation will only lead to more overhead instead of benefit. For the remaining applications, we need to aggregate 7 (SD) to 230 (Sharepoint) clients to identify all the TP's.

7.4 Performance & Scalability

In this section, we focus on the performance and scalability of Orion. We are interested in answering the following questions: i) does it scale with the number of services in the network? ii) what is the CPU and memory usage in a network-based or a host-based deployment? iii) how quickly can dependencies be extracted when administrators need such information?

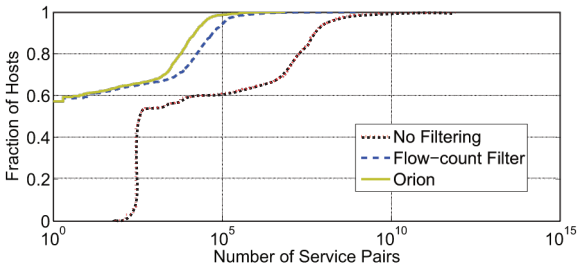


Figure 15: *Flow-count and time-window filters improve scalability*

7.4.1 Scalability of delay distribution calculator

As described in Section 5.2, Orion uses a time window of three seconds and a flow count threshold of 50 to filter unnecessary service pairs. To illustrate the effectiveness of these two heuristics, Figure 15 plots the CDF of the number of service pairs without any filter, only with the flow count filter, and with both filters. The X-axis is the number of service pairs and the Y-axis is the cumulative fraction of hosts. The flow count filter reduces the service pairs by almost three orders of magnitude. After applying the time-window filter, 99% of the hosts have fewer than 10^5 service pairs. As we show next, the actual memory usage is reasonably small for both the network-based and the host-based deployment.

7.4.2 Performance of flow generator & delay distribution calculator

As shown in Figure 4, we currently use the network-based deployment by running Orion on dedicated sniffing boxes attached to three routers in the network. In this deployment, each sniffing box may capture large volumes of traffic from multiple hosts in the same subnet. We want to understand whether the flow generator and delay distribution calculator can keep up with such high traffic rate. We use the traffic at R_0 for our evaluation because it contains the aggregate traffic of all the clients and is bigger than the traffic at the other two routers. We run the flow generator and delay distribution calculator on a Windows Server 2003 machine with 2.4G four-core Xeon processor and 3GB memory. We measured their performance during the peak hour (2 - 3 PM local time) on a Thursday. The aggregate traffic rate is 202 Mbps during that period. The flow generator processed one hour of traffic in only 5 minutes with an 8MB memory footprint. The delay distribution calculator finished in 83 seconds and used 6.7MB memory.

We also measure the performance of the host-based deployment. Given that Orion has to share resources with other services on the same host, we focus on its CPU and memory usage. We perform the evaluation on a regular client machine with 3GHz Pentium4 processor and 1GB memory. The packet sniffer, flow generator, and delay distribution calculator normally use 1% of CPU and 11MB memory in total. Such overhead is reasonably small for long-term monitoring on individual hosts.

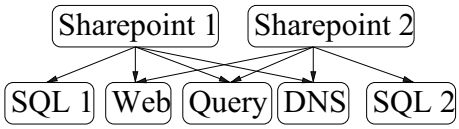


Figure 16: *Using dependency graph for fault localization*

7.4.3 Performance of dependency extractor

We now evaluate the execution time for extracting dependencies from the delay distributions. The results are measured on a Windows Server 2003 machine with 2G dual-core Opteron processor and 4GB memory. For all the applications, the dependency extraction was finished within two minutes. This is short enough for administrators to run on-demand. We also measured the network usage of client aggregation. Aggregation is required for merging the delay distributions from multiple clients when there are insufficient delay samples. During the two-week evaluation period, the total size of the delay distributions from all the 2,048 clients is under 1MB after compression. This suggests it is feasible to use a centralized dependency extractor since it is unlikely to become the bottleneck.

8 Operational Use of Dependencies

We now provide examples of how dependencies can facilitate fault localization and reconfiguration planning. These examples are by no means exhaustive. Administrators may find dependencies useful for other network management tasks, *e.g.,* impact analysis and anomaly detection.

The existence of complex dependencies between different services makes it extremely challenging to localize sources of performance faults in large enterprise networks. For instance, when a Sharepoint service fails, it could be caused by problems at the DNS servers, SQL servers, web servers, or query servers. Manually investigating all these relevant servers for each performance fault is time-consuming and often infeasible.

A dependency graph summarizes all the components that are involved in particular services. Combined with observations from multiple services, it enables fast and accurate fault localization. Figure 16 illustrates an example of a dependency graph with two Sharepoint services. For simplicity, we ignore problems in the network and the port numbers in the graph. $Sharepoint_1$ and $Sharepoint_2$ use the same DNS, query, and web servers. However, they use different SQL servers for storing contents. Suppose $Sharepoint_1$ is experiencing problems while $Sharepoint_2$ is not. From the dependency graph, we deduce that the source of the problem is unlikely at

the DNS, query, or web servers since $Sharepoint_2$ has no problems using them. This leaves SQL_1 as the most plausible candidate for the source of the problem.

While the above example is fairly simple, a dependency graph of a large enterprise network will be substantially more complex. In fact, it is almost impossible to inspect manually. Fortunately, there have been known techniques that automate the fault localization process by applying Bayesian inference algorithms to dependency graphs [8, 20, 19]. We omit the details here since they are not the focus of this paper.

Another use of dependencies is in reconfiguration planning. Large enterprises have many services and applications, which are continually being reorganized, consolidated, and upgraded. Such reconfigurations may lead to unanticipated consequences which are difficult to diagnose and repair. A classic example involves a machine configured as a backup database. Since there is no explicit documentation about this dependency, the machine is recycled by administrators. Later on, when the primary database fails, applications that depend on the database becomes completely unavailable.

To avoid such unanticipated consequences, administrators must identify the services and applications that depend on a particular service before any changes can be made to that service. This often is a slow and expensive process. Given the dependencies extracted from all the service pairs, we can easily search for all the services that directly or indirectly depend on a particular service. This will significantly save the time administrators spend in assessing and planning for the changes.

Besides our own research prototype, a production implementation of Orion based on the TDI driver is currently being deployed in the Microsoft IT department (MSIT). The administrators will initially use the dependencies extracted by Orion for reconfiguration planning. Orion has been set up on two web services and one Windows Messenger service. Preliminary results indicate Orion has successfully discovered the set of expected dependencies on database servers, AD servers, and presence servers. The plan is to roll out Orion to over 1,000 services managed by MSIT in the next six months.

9 Conclusion

In this paper, we present a comprehensive study of the performance and limitations of dependency discovery techniques based on traffic patterns. We introduce the Orion system that discovers dependencies for enterprise applications by using packet headers and timing information. Our key observation is the delay distribution between dependent services often exhibits "typical" spikes that reflect the underlying delay for using or providing such services. By deploying Orion in Microsoft's corporate network that covers over 2,000 hosts, we extract the dependencies for five dominant applications. Our results from extensive experiments show Orion improves the state of the art significantly. Orion provides a solid foundation for combining automated discovery with simple testing to obtain accurate dependencies.

References

[1] EMC SMARTS. http://www.emc.com/products/family/smarts-family.htm.

[2] HP OpenView. http://www.openview.hp.com.

[3] IBM Tivoli. http://www.ibm.com/software/tivoli/.

[4] Mercury MAM. http://www.mercury.com/us/products/business-availability-center/applica%tion-mapping.

[5] Microsoft MOM. http://technet.microsoft.com/en-us/opsmgr/bb498230.aspx.

[6] Taming Technology Sprawl. http://online.wsj.com/article/SB120156419453723637.html.html?mod=techno%logy_main_promo_left.

[7] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of SOSP*, 2003.

[8] 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. In *Proc. ACM SIGCOMM*, 2007.

[9] P. V. Bahl, P. Barham, R. Black, R. Chandra, M. Goldszmidt, R. Isaacs, S. Kandula, L. Li, J. MacCormick, D. Maltz, R. Mortier, M. Wawrzoniak, and M. Zhang. Discovering Dependencies for Network Management. In *HotNets*, 2006.

[10] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modeling. In *OSDI*, 2004.

[11] G. Bartlett, J. Heidemann, and C. Papadopoulos. Understanding passive and active service discovery. In *IMC*, 2007.

[12] R. Black, A. Donnelly, and C. Fournet. Ethernet Topology Discovery without Network Assistance. In *ICNP*, 2004.

[13] O. E. Brigham. The fast fourier transform and its application. In *Prentice-Hall*, 1988.

[14] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Intergrated Network Management*, 2001.

[15] M. Chen, A. Accardi, E. Kcman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based Failure and Evolution Management. In *NSDI*, 2004.

[16] R. Fonseca, G. Porter, R. H. Katz, S. Shenkar, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. In *NSDI*, 2007.

[17] J. F. Kaiser and R. W. Schafer. On the Use of the Io-Sinh Window for Spectrum Analysis. In *IEEE Transactions on Acoustics, Speech and Signal Processing*, 1980.

[18] S. Kandula, R. Chandra, and D. Katabi. What's Going On? Extracting Communication Rules In Edge Networks. In *Proc. ACM SIGCOMM*, 2008.

[19] S. Kandula, D. Katabi, and J. P. Vasseur. Shrink: A Tool for Failure Diagnosis in IP Networks. In *MineNet*, 2005.

[20] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. IP Fault Localization via Risk Modeling. In *NSDI*, 2005.

[21] B. Lowekamp, D. R. O'Hallaron, and T. R. Gross. Topology Discovery for Large Enternet Networks. In *SIGCOMM*, 2001.

[22] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. WAP5: black-box performance debugging for wide-area systems. In *WWW*, 2006.

SQCK: A Declarative File System Checker

Haryadi S. Gunawi, Abhishek Rajimwale,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
Computer Sciences Department, University of Wisconsin-Madison

Abstract

The lowly state of the art for file system checking and repair does not match what is needed to keep important data available for users. Current file system checkers, such as e2fsck, are complex pieces of imperfect code written in low-level languages. We introduce SQCK, a file system checker based on a declarative query language; declarative queries are a natural match for the cross-checking that must be performed across the many structures of a file system image. We show that SQCK is able to perform the same functionality as e2fsck with surprisingly elegant and compact queries. We also show that SQCK can easily perform more useful repairs than e2fsck by combining information available across the file system. Finally, our prototype implementation of SQCK achieves this improved functionality with comparable performance to e2fsck.

1 Introduction

Access to data is critical for both business and personal users of computer systems. Data is often either priceless or very expensive to re-obtain if lost; downtime and data loss combine to cost companies and end-users billions of dollars each year [22, 29]. As the central repository for much of the world's data, file systems play a central role in data protection and management. Thus, file systems should be robust and reliable.

A key component to a robust file system is a robust offline file system checker. Tools such as fsck have existed for many years [24] and are applied to restore a damaged or otherwise inconsistent file system image to a working and usable state. Although many newer file systems have tried to avoid the inclusion of an offline checker in their tool suite [19] (for example, by assuming that journaling always keeps the file system consistent), they inevitably find that a checker must be deployed. For example, SGI's XFS was introduced as a file system with "no need for fsck, ever," but soon found it necessary to deliver such a tool [15].

Unfortunately, robust checkers are not currently straightforward to design or implement. First, checkers are large and complex beasts; for example, the Linux ext2 checker contains more than thirty thousand lines of C code, while the ext2 file system itself is less than ten thousand lines. Checkers are often written in a low-level systems language such as C, which can be difficult to reason about. Checkers also are hard to test, given the huge possible state space of input file systems. Finally, checkers are often run only when a serious problem has occurred; it is well known that rarely-run recovery code tends to be less reliable [9, 28].

Given these realities, it is perhaps not surprising that file system checkers often corrupt or lose data [31, 32]. Recent work in model checking has found many serious implementation flaws in checkers, including invalid write ordering, buggy transaction abort, incorrect optimization, and unattempted recovery of invalid directory entries [31, 32]. Our evaluation of e2fsck, the Linux ext2/3 checker, confirms that it has many weaknesses. In particular, e2fsck sometimes performs inconsistent repairs that can corrupt the file system image by overwriting important metadata (including the superblock); e2fsck also sometimes does not use all available information and can lose portions of the directory tree.

To build a new generation of robust and reliable file system checkers, we believe a new approach is required. The ideal approach should enable the high-level intent of the checker to be specified in a clear and compact manner; further, the description of the intent should be cleanly separated from its low-level implementation and how it is optimized. A high-level specification has multiple benefits: by its very nature it is easier to understand, modify, and maintain.

In this paper, we introduce SQCK (pronounced "squeak"), a novel file system checker. Borrowing heavily from the database community, SQCK employs declarative queries to check and repair a file system image. We find that a declarative query language is an excellent match for the cross-checks that must be made across the different structures of a file system.

Our experience shows that declarative repairs can be surprisingly elegant and compact, especially compared to the original e2fsck code. Specifically, we find that SQCK can reproduce the functionality of e2fsck in many fewer lines of code; the SQCK checks and repairs require only about 1100 lines of SQL (along with some helper code written in C).

We find that SQCK can improve upon the traditional checks and repairs as well. First, SQCK avoids the inconsistent repairs performed by e2fsck by ensuring that its queries are executed in the correct order; specifically, a file system structure is only repaired after the location of that structure has been validated. Second, SQCK can perform more interesting and complete repairs than e2fsck by combining information from multiple sources. For example, SQCK performs majority voting over superblock and group descriptor replicas to handle the case where the primary copy is corrupted. SQCK also examines the ".." entry of a directory to verify the correct parent when there is conflicting information. Finally, SQCK ensures that its repairs follow the same allocation policies as ext2 by laying out new blocks with the appropriate locality.

SQCK achieves this simplicity and completeness with no cost to performance. Our evaluation of the first-generation prototype of SQCK on top of the MySQL DBMS [1] shows that SQCK can handle even large file system partitions with comparable performance to e2fsck. Overall, we believe that the SQCK-style declarative approach will lead to a new generation of simpler, more robust, and more complete file system checking and repair.

The rest of this paper is organized as follows. In Section 2, we present background information on the state of the art of checking and evaluate a traditional file system checker. We present the design and implementation of SQCK in Sections 3 and 4 and then evaluate SQCK in Section 5. We then discuss related work in Section 6 and conclude in Section 7.

2 Fsck Background

To create a better file system checker, one first needs to understand the current state of the art. In this section, we give a brief overview of the checks and repairs performed by e2fsck for an ext2 file system [10]. We then describe in detail the weaknesses and non-optimal repairs performed by e2fsck. Finally, we explain why modeling languages [11, 12, 21] are not as suitable as declarative query languages for file system checking.

2.1 Fsck Overview

Despite the best efforts of the file and storage system community, file system images become corrupt and require repair. While it is obvious that non-journaling file systems (*e.g.*, ext2) can easily become inconsistent due to untimely crashes, other file systems can as well. In particular, problems with many different parts of the file and storage system stack can corrupt a file system image: disk media, mechanical components, drive firmware, the transport layer, bus controller, and OS drivers [6, 7, 17, 18, 27, 30]. Since file systems do not usually contain the machinery to fix corruptions themselves [8, 27], there is a broad need for robust file system checkers.

Given both its popularity and our ability to access its source code, we focus on the file system checker for ext2/ext3, e2fsck. The purpose of the e2fsck utility is to check and repair the data structures of an ext2/ext3 file system on disk; in the ideal case, the repaired file system is readable, writable, and contains all of the directories, files, and data of the original file system.

e2fsck is a non-trivial piece of code: it contains more than 30,000 lines of C code and can identify and return 269 different error codes. Its checks and repairs are performed in six different phases [24], in which scanning the disk, checking the data structures, and repairing any inconsistencies are all intermixed. Many of the simplest checks examine individual structures in isolation (*e.g.*, that superblock fields, inode fields, and directory entries all appear valid) or verify that pointers fall within the expected ranges. More interesting and costly checks validate that no two pointers (*e.g.*, across all inodes) point to the same data block. Other intensive checks peruse the file system tree, ensuring that all files and directories are properly connected.

2.2 Fsck Weaknesses

To understand the weaknesses of e2fsck, we need to understand the individual repairs performed by e2fsck in response to different errors it encounters. We are not explicitly interested in finding implementation bugs [31, 32], but in understanding when e2fsck could have made better repairs than it did for a given corruption.

2.2.1 Fault Injection Methodology

To begin to understand the complex runtime behavior of e2fsck, we explore how e2fsck repairs a single on-disk corruption. Given that it is infeasible to exhaustively corrupt every data structure field to every possible value, we limit our scope to corrupting on-disk pointers. Ext2 contains two classes of pointer. First, a block pointer contains a physical block number; for example, data block pointers in inodes contain the block numbers of corresponding data blocks. Second, an index pointer contains an index into a table; for example, an inode index picks an entry in the inode table within a block group.

We use our knowledge of ext2 to further reduce the search space. In particular, we corrupt pointers in a type- and location-aware fashion [8]. Specifically, we assume that the e2fsck repair depends only on: (i) the type of pointer that has been corrupted, and (ii) the type of block that it points to after corruption and whether it lives in the same or a different block group. For example, (i) corrupting File A's data pointer is the same as corrupting

File B's data pointer, and (ii) corrupting a pointer to refer to inode-block P in group G is the same as corrupting it to refer to inode-block Q in group G.

To corrupt the file system and examine the results, we use the debugfs utility [2]. We corrupted approximately 10 different pointers to 18 different values for a total of 180 corruption tests. To the best of our knowledge, all of our findings are new.

2.3 Results

From our fault injection experiments, we find that e2fsck fails along four different axes. First, e2fsck does not always create a *consistent* file system, even though this is the explicit purpose and goal of fsck. In fact, in some cases, e2fsck will perform an imprudent "repair" that transforms a file system with a relatively small inconsistency into one that is completely unreadable.

Second, e2fsck does not always perform an *information-complete* repair. We define a repair to be information-complete if it reconstructs the file system to match the original file system to the greatest extent possible given the information available on disk. The notion of an information-complete repair is needed because a repair can easily create a consistent, but useless file system by simply removing all of the contents. For example, an information-complete repair should always incorporate redundant copies.

Third, e2fsck does not always perform a *policy-consistent* repair. We define a repair to be policy consistent if it follows the same policies as the original file system; for example, since ext2 allocates data blocks in the same group as its corresponding inode, e2fsck should as well.

Finally, e2fsck does not always perform a *secure* repair. Specifically, e2fsck sometimes leaks information from one data structure to another when it clones blocks. In this way, it is possible for a user's file to be "repaired" to contain data from a file in the root directory.

We now describe the specific behavior of e2fsck that leads to these problems.

Inconsistent Repair: *Clears "Indirect Blocks" Incorrectly.* Fundamentally, e2fsck checks and repairs certain pointers in an incorrect order; as a result, e2fsck can itself corrupt arbitrary data on disk, even the superblock. Specifically, e2fsck clears block pointers that fall out of range of the file system inside indirect blocks without first checking that the pointer to the indirect block itself is correct. Thus, if an indirect pointer was corrupt, e2fsck may clear the block that the indirect pointer incorrectly refers to. This clearing can lead to arbitrary corruptions of file, directory, and meta-data in the file system; most notably, if the file system contains only a single superblock, the file system can even be unmountable after running e2fsck.

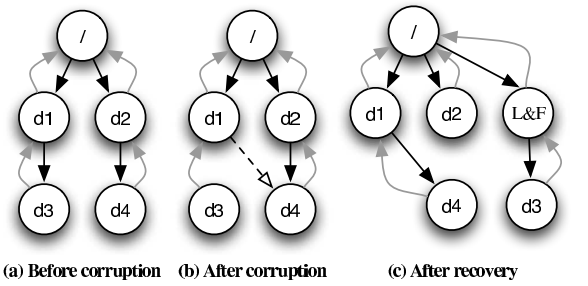


Figure 1: **The false parenthood problem.** *This figure shows the problem in the recovery done by e2fsck for corruption in directories. Each node is a directory in the file system. For clear understanding, we use dotted backpointers to show the parent for each directory as present in the '..' entry for that directory. Part (a) of the figure shows the initial file system structure. Part (b) shows the file system structure after corruption. We inject this fault where the entry for dir3 in dir1 is corrupted to point to the inode of dir4. After recovery by e2fsck, the dir1 claims dir4 and the original parent child link between dir2 and dir4 is deleted. This results in totally different structure of the file system after recovery as shown in part (c). For convenience we show the lost+found (L&F) directory only in the final structure.*

Information Incomplete: *False Parenthood.* e2fsck does not always use all of the information available to it regarding directories. One example is the case where an inode index within a directory is corrupted to point to a different valid directory inode. This situation is illustrated in Figure 1. If a directory entry is corrupted to point to another target directory (parts a and b), the e2fsck repair might move the target directory to the wrong parent (part c).

We emphasize that enough information is available in an ext2 file system for e2fsck to make the correct repair: each directory contains an entry for its parent (denoted ".."). To perform an information complete repair, e2fsck could simply observe this entry to keep the target directory with its correct parent and to reattach the lost directory to its parent instead of moving it to lost+found. In general, the directory hierarchy in ext2 contains much more information than is being used currently in e2fsck.

Information Incomplete: *Ignores Replicas of Inode Table Pointers.* Ext2 contains replicas for important meta-data, such as pointers to the inode tables; however, e2fsck does not always use this redundant information. For example, when an inode table pointer becomes corrupted and points to other blocks inside the same block group, e2fsck assumes the pointer is correct; e2fsck then finds that the "inodes" are not valid. For consistency, e2fsck removes the corresponding directories and files from the directory tree; if this group contains the root directory, the file system is trivially consistent with no directories. Again, enough information is available for e2fsck to make the correct repair: each inode

table pointer is replicated across block groups; e2fsck should check that all block groups agree on these important values.

Policy Inconsistent: *Different Layout.* e2fsck does not allocate blocks on disk with the same layout policy as ext2; as a result, e2fsck can fragment files and directories, degrading the future performance of file system operations. For example, when e2fsck detects that the same data block is pointed to by both a directory and a file, e2fsck clones the block by allocating a new block for the file and retaining the old block for the directory. To perform a policy-consistent repair, e2fsck should allow the closer inode to retain the original data block.

Insecure Repair: *Copies Data Freely.* Whenever e2fsck discovers that two pointers refer to the same block, e2fsck clones the block. However, this policy has the potential to leak private information. For example, if a data block is shared by two inodes, one in the `/home/userA` directory and one in the `/root` directory, we might want to remove the pointer from `userA` and keep the one from the root.

Summary: We have found that e2fsck has a number of problems in how it performs repairs; we note that these problems are not simple implementation bugs, but are fundamental design flaws. In particular, it is difficult for e2fsck to combine the many pieces of information available (*e.g.*, replicas of pointers and parent directory entries) and to ensure that all checks and repairs are done in the correct order.

2.4 Other Approaches

Given the difficulties of implementing a file system checker, an alternative is needed. File system checking can be viewed as ensuring that the content satisfies a specification; therefore, some researchers have attempted to auto-generate fsck code by writing a specification in an object modeling language. Specifically, Demsky and Rinard's work repairs inconsistencies automatically given specified constraints [12]. Their automated repair finds the cheapest way to repair the system such that it satisfies the constraints again. For example, if two inodes share the same data block, the cheapest repair could simply remove one of the pointers; however, this may not be the desired result. In fact, there are many ways to solve the problem: the inode with the earliest modification time could release the block [24], the block could be cloned (e2fsck's way), or the operator could decide. In our terminology introduced above, previous approaches ensure that the repairs are consistent, but not necessarily information-complete or policy-consistent.

When reinventing fsck, we need a language that can declaratively express both the checks and the repairs. Like others who have applied declarative languages to domains such as system configuration [13] and network overlays [23], we believe that the solution is to use a declarative query language. Declarative query languages have been built from day one to both cross-check and update massive amount of data. Hence, we believe utilizing a declarative language is a better fit than a specific modeling language for fsck.

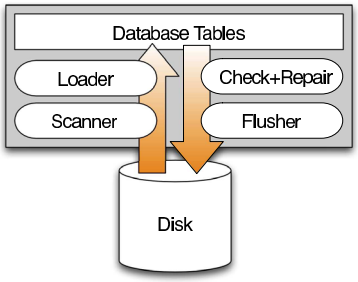


Figure 2: **Architecture.** *The diagram depicts the basic SQCK architecture. The left part of the design, the loader and scanner, and the right part of the design, the checker and flusher are decoupled, allowing us to optimize each component in isolation.*

3 Designing SQCK

In this section, we describe the design of SQCK, our file system checker based on declarative queries. We first describe our goals and then present the overall architecture of SQCK, including how declarative checks and declarative repairs are performed. We then describe three simple versions of SQCK: one that emulates e2fsck, one that fixes the inconsistent repairs of e2fsck, and one that significantly improves the types of repairs performed.

3.1 Goals

We believe that a file system checker should be correct, flexible, and have reasonable performance; we believe a declarative language will enable us to meet these goals for the following reasons.

Correctness: The primary responsibility of a file system checker is to produce a consistent file system image. A declarative language allows one to check and repair hundreds of corruption scenarios in a clean and compact fashion; we believe the ability to produce correct repairs is improved due to the simplicity of the queries and the separation of the specification from the implementation. A secondary goal is to produce repairs that leverage all of the on-disk information to retain as much as possible of the file system. We believe declarative languages allow one to easily combine the disparate information that resides throughout the file system.

Flexibility: Given a single corruption, there are many reasonable repairs that could be performed. The simplicity of a declarative language encourages one to explore

Tables	Fields
Superblock Table	*blkNum*, *copyNum*, *dirty*, firstBlk, lastBlk, blockSize, ...
GroupDesc Table	*blkNum*, *gdNum*, *copyNum*, *dirty*, *start*, *end*, blkBitmap, inoBitmap, iTable, ...
Inode Table	*ino*, *blkNum*, *used*, *dirty*, mode, linksCount, blocksCount, size, ...
DirEntry Table	*blkNum*, *entryNum*, *dirty*, ino, entryIno, recLen, nameLen, name
Extent Table	start, end, *pBlk*, *pByte*, *type*, *startLogical*, *endLogical*, *ino*, *dirty*, ...

Table 1: **SQL Tables.** *Italic fields represent information we generate since they are not stored on the disk.*

this policy space and even provide different modes of repair (*e.g.*, fast but partial repair, or slow but full/smart repairs).

Performance: While the performance of a file system checker is not a primary concern, it must not be prohibitively slow; specifically, the checker must be able to handle the amount of data on modern disks and storage systems. Thus, our goal is to create a checker that is competitive in speed to the original e2fsck.

3.2 Architecture

SQCK contains five primary components, as shown in Figure 2. The *scanner* reads the relevant portions of the file system from the disk, while the *loader* loads the corresponding information into the database *tables*. The *checker* is then responsible for running the declarative queries that both check and repair the file system structures. The *flusher* completes the loop by writing out the changes to disk. We postpone our description of the scanner, loader, and flusher until Section 4. In this section, we explain the tables and the checker.

3.2.1 Database Tables

It is important to construct the database tables such that the SQCK checker can perform efficient queries that cover the same repairs as e2fsck. Conceptually, SQCK contains a table for each of the different metadata types in the file system: superblocks, group descriptors, inodes, directories, and block pointers [10]. Together, the tables store all of the information about the file system image that was originally on disk. However, with this on-disk information alone, the SQCK checks and repairs are neither simple nor efficient; therefore, SQCK stores extra, easily calculated information in the tables. Table 1 shows the five database tables utilized by SQCK. We describe briefly the important fields in each table.

Superblock: Since the superblock is replicated, we load each replica into a row of the table; this table allows SQCK to easily check the consistency across superblocks. As expected, each row contains the information available from the superblocks on disk. To be able to reflect repairs back to the disk in the flusher, we also introduce *copyNum* and *blkNum* fields that specify where a replica lives on the disk and a *dirty* field.

GroupDescTable: Each group descriptor and its replicas are loaded into separate rows of this table; as expected, we store here the on-disk information such as the pointers to the block bitmap, inode bitmap, and inode table. SQCK also adds the *start* and *end* block of each group; this addition allows SQCK to easily check whether pointers fall within the desired range of the block group.

InodeTable: Each row of the table corresponds to a different allocated inode, with appropriate fields for the on-disk information such as mode, links, and size. The *used* field tracks which inodes are part of the final directory tree so that SQCK can calculate the final inode bitmap.

DirEntryTable: Each row of the table corresponds to a different directory entry. SQCK performs many cross-checks on this table to verify the directory tree structure.

ExtentTable: The conceptual idea of this table is to record all of the pointers to data blocks, so that SQCK can ensure that no two pointers refer to the same block. In our initial implementation, we loaded each direct pointer as its own row; however, this is intractable for a large file system because the table grows too large and the loader takes too long. Therefore, we switched our table design to represent extents of contiguous direct blocks; specifically, each extent specifies the start and end block. Additionally, each row records the location of the original pointer and the *type* of the pointer (*e.g.*, direct, single, double, or triple indirect).

3.2.2 Declarative Checks

A declarative query language is an excellent match for the checks and repairs that must be performed by a file system checker. To give some intuition as to why this is true, we categorize the different checks that must be made and show how a prototypical check from each category can be specified with SQL [3].

The original e2fsck performs a total of 121 interesting repairs. We have categorized all of these repairs into four categories, depending upon how many file system structures the repair must simultaneously peruse. As shown in Table 2, a repair can touch a single instance of a single structure type, one instance of one type with another of a different type, multiple instances of the same type, or multiple instances from multiple types.

There are 63 fsck repairs that involve fields of a single structure in isolation. A simple example of this type of repair is ensuring that the deletion time of a used inode is zero. Another example is verifying that the block bitmap for a group is located within that group. We show

```
SELECT *
FROM   GroupDescTable G
WHERE  G.blkBitmap NOT BETWEEN G.start AND G.end
```

Figure 3: **Check block bitmap not in group.** *This query finds a block bitmap pointer (blockBitmap) of a group that points outside the group.*

```
SELECT  X.*
FROM    ExtentTable X, SuperblockTable S
WHERE   S.copyNum =  1                  AND
        X.type    =  INDIRECT_POINTER   AND
       (X.start   <  S.firstBlk         OR
        X.end     >= S.lastBlk)
```

Figure 4: **Check illegal indirect block.** *An illegal indirect block is one that points to outside the file system range*

```
SELECT  *
FROM    DirEntryTable P, DirEntryTable C
WHERE   // P says C is his child
        P.entryNum >= 3      AND
        P.entryIno  = C.ino  AND
        // but C says P is not his parent
        C.entryNum  = 2      AND
        C.entryIno <> P.ino
```

Figure 5: **Bad dot dot.** *This query finds a directory entry that does not claim the actual parent.*

```
SELECT X.*
FROM   ExtentTable X
WHERE EXISTS
 (SELECT *
  FROM   SuperblockTable S
  WHERE
   // extent overlaps superblock copies
   S.blk BETWEEN X.start AND X.end)
 OR EXISTS
 (SELECT *
  FROM   GroupDescTable G, SuperblockTable S
  WHERE
   // or extent overlaps group descriptors
   (X.start BETWEEN G.blk AND G.blkEnd OR
    X.end   BETWEEN G.blk AND G.blkEnd) OR
   // or extent overlaps inode table
   (X.start BETWEEN G.iTbl AND G.iTblEnd OR
    X.end   BETWEEN G.iTbl AND G.iTblEnd) OR
   // or extent overlaps block bitmap
    G.blkBitmap BETWEEN X.start AND X.end OR
   // or extent overlaps inode bitmap
    G.inoBitmap BETWEEN X.start AND X.end)
```

Figure 6: **Check block overlaps metadata.** *This query locates inode's extents that overlap with the filesystem metadata. To reduce space, we abbreviate some fields: G.iTblEnd should be G.iTable + S.inodeBlocksPerGroup - 1; G.blkEnd should be G.blk + S.gdBlks - 1.*

	Single instance	**Multiple instances**
Intra structure	*Category #1* 63 checks	*Category #3* 11 checks
Inter structures	*Category #2* 12 checks	*Category #4* 35 checks

Table 2: **Taxonomy of fsck cross-checking.** *We distinguish four types of cross-check. We report the number of checks that fall into each category. In the first category, a cross-check can be made within an instance of a structure. In the second, a cross-check is performed on an instance of a structure and an instance of another different structure. The third category cross-checks multiple instances of a structure. Finally, the last category involves information stored in multiple instances of more than one structures. Each number in the box represents the number of checks that are done by e2fsck in each category.*

how this check can be expressed simply and efficiently using SQL in Figure 3. The query simply performs a SELECT from the group descriptor table to find any bitmaps that are not within the desired range for the group. Thus, range-checking queries are easily specified.

The second category includes checks between one instance of a structure and an instance of another different structure; fsck runs 12 checks of this type. A simple example is verifying that all pointers refer to blocks within the file system; this check involves verifying that every pointer is within the range specified in the primary superblock. Unlike the previous example, this example must examine values in different structures and subsequently different tables. Figure 4 shows how to check that no indirect block points outside the file system. Specifically, the query returns all extents (`X.start..X.end`) corresponding to indirect pointers that fall outside the file system range specified in the primary superblock (`S.firstBlk..S.lastBlk`). Hence, SQCK can easily join multiple structures to perform the necessary cross-checks.

The third category contains 11 cross-checks of multiple instances of the same structure. One example of this type of repair is checking that multiple inodes do not point to the same data block. A second example, shown in Figure 5 checks that the ".." entry of a directory points to the actual parent. This check can be done easily in SQL: the query simply joins the directory entry table with itself, selecting cases where the parent directory contains an entry for a child (where `P.entryNum >= 3`), but the child's entry for ".." (`P.entryNum = 2`) is not the parent's inode.

Finally, 35 checks fall into the fourth category in which the cross-checks involve multiple instances of more than one structure. One example is the rule that validates the link count of an inode, since it must traverse all directory entries and count how many times each en-

```
SELECT  P.entryIno, COUNT(*), MIN(P.ino)
FROM    DirEntryTable P, InodeTable I
WHERE   P.entryNum >= 3      AND
        P.entryIno  = I.ino  AND
        I.mode      = DIR
GROUP BY P.entryIno
HAVING (COUNT(P.entryIno) > 1)
```

Figure 7: **Check multiple parents.** *This query returns directories that have multiple parents. The parent that has the smallest inode number (*`MIN(P.ino)`*) will be the one that keeps the child directory.*

```
UPDATE  ExtentTable X
        INNER JOIN
        (Query in Figure 4) AS V
        ON X.ino   = V.ino    AND
           X.type  = V.type   AND
           X.start = V.start  AND
           X.end   = V.end
SET     X.start = 0, X.end = 0, X.dirty = 1
```

Figure 8: **Repair illegal indirect block number.** *This query repairs indirect block numbers that fall outside the file system range (returned by query in Figure 4), by clearing them to zero.*

```
result = run(findUnconnectedDir.sql);       [9]
while(dir = mysql_fetch_row(result)) {
 run(changeDotDot.sql, dir, lfIno);         [3]
 slot = run(findEntrySlot.sql, lfIno);      [7]
 if (!slot) {
  lfBlk = run(getLocation.sql, lfIno);      [3]
  newBlk = run(allocNewBlock.sql, lfBlk); [25]
  if (run(needIndirect.sql, lfIno))         [5]
   { // alloc indirect (not shown) }
  run(addNewBlock.sql, newBlk, lfIno);      [3]
  run(addInodeSize.sql, lfIno);             [3]
  run(initNewDirBlk.sql, newBlk, lfIno);    [3]
  slot = run(findEntrySlot.sql, lfIno);     [7]
 }
 // now break the slot and prepare         [13]
 // newSlot based on dir.  (not shown)
 run(updateOldSlot.sql, oldSlot);           [3]
 run(insertNewSlot.sql, newSlot);           [3]
 run(incrementLinkCount.sql, lfIno);        [3]
}
```

Figure 9: **Complex repair.** *The C pseudo-code above illustrates the complex repair in reattaching unconnected directories to the lost+found directory. The bold texts are the SQL files that are executed. The bold numbers in the brackets represent the lines count of each SQL file. The italic number is the lines count of the C code.* `lfIno` *is the inode number of the lost+found directory.*

try appears. We give two examples of these queries to further convince the reader that even these types of seemingly complicated checks are surprisingly straightforward to express.

The first example checks for conflicting block pointers; in ext2, block pointers are stored in many places and none should refer to the same block. Figure 6 shows a query that ensures blocks pointed from an inode do not overlap with file system metadata blocks. The query is a little bit cumbersome because it checks whether an extent overlaps with each piece of file system metadata separately (*i.e.*, superblock copies, group descriptors, inode bitmaps, block bitmaps, and inode tables).

The second example verifies that multiple directory entries do not point to a same directory, corresponding to the false parenthood problem discussed in Section 2.3; we show how it can be expressed in SQL in Figure 7. Basically, the query selects directory entries that appear more than once in the tree structure. In more detail, the query does not select the "." or ".." entries and selects only directory inodes, as determined by their `mode` field in the inode table. Counting the number of entries satisfying this constraint is straightforward with the ORDER BY and HAVING features of the query language. Note that this query returns the smallest inode number among the parents (`MIN(P.ino)`), which is needed to mimic how e2fsck incorrectly repairs this problem. In particular, e2fsck always assumes the parent with the smallest inode number is the real parent without consulting the ".." entry of the child. We show how we can easily improve this query in Section 3.3.3.

3.2.3 Declarative Repairs

Performing checks of file system state is only part of the problem; after SQCK detects an inconsistency, it must then perform the actual repair. SQCK performs the repair by first modifying its own tables; the flush process then propagates these changes to the disk itself. We have found that repair operations on the tables can be performed in one of two ways.

In the simplest cases, a repair must simply adjust a few fields within a table. These repairs can be performed by embedding the declarative checks presented previously into a larger query that then sets fields within the selected rows. For example, an illegal indirect block pointer (one that points outside the file system range) is fixed by clearing the pointer to zero. Figure 8 shows that these pointers can be cleared with a query that sets to zero the illegal extents found by the check query in Figure 4. Note that the query also sets the dirty flag so that the flusher will later propagate these changes from the database tables to the on-disk structures.

In more complex cases, repairs may need to update more than one table. In these cases, SQCK combines a

series of SQL queries with C code. SQCK currently supports a variety of repair primitives, such as finding free blocks and inodes and adding and deleting extents, directory entries, and inodes. Figure 9 shows how a valid directory with a reference count of zero (*i.e.*, a lost directory) is reconnected to the lost+found directory. Briefly, the code behaves as follows. After a query finds the set of unconnected directories, SQCK performs the following operations on each such directory. First, the ".." entry is adjusted to point to lost+found. Next, a directory entry slot is allocated within lost+found, which may require allocating new blocks and increasing the size of the lost+found. After the slot is ready, the entry is filled to correspond to the unconnected directory.

3.3 Possible Repair Policies

The simplicity of implementing checks and repairs in SQCK enables one to construct different versions with different repair policies. At this time, we have created three versions of SQCK: one that emulates e2fsck with both its good and bad polices, one that fixes what e2fsck does wrong (*i.e.*, fixes the inconsistent repairs described in Section 2), and one that adds new functionality that e2fsck does not even attempt (*i.e.*, performs information-complete and policy-consistent repairs). We briefly describe these three different versions.

3.3.1 Emulating e2fsck

Our basic version, SQCK$_{fsck}$, emulates the repairs made by e2fsck. From our analysis of e2fsck, we have determined that it performs 153 different repairs, of which 121 are significant and interesting for ext2 (the remaining 32 repairs fix the ext3 journal and other optional features). These 121 repairs are detailed in Table 3. As shown, e2fsck performs these repairs in six distinct phases, in which reading the file system image from the disk is intermixed with the actual checks and repairs. SQCK$_{fsck}$ implements these 121 repairs each as a separate query within the check and repair process.

3.3.2 Fixing e2fsck

Our second version, SQCK$_{correct}$, fixes the inconsistent repairs performed by e2fsck. As described in Section 2, if e2fsck follows a bad pointer to what it believes is an indirect block, it can corrupt the file system and leave it in an inconsistent state. The basic flaw of e2fsck is that it performs certain checks and repairs in the incorrect order: it wrongly "repairs" direct pointers before checking that the indirect block containing those pointers is valid.

In general, repairs of a complex data structure must be performed in a specific order; specifically, if a piece of information A is obtained from B, then B must be checked and repaired first.

#	Checks Performed
28	**Phase 0: Check consistency in the superblock**
23	*Field check:* Check all superblock fields (*e.g.*, fs size, inode count, groups count, mount/write time)
3	*Range check:* Ensure pointers to block bitmap, inode bitmap, inode table are in the group
2	*Special feature:* Check resize inode feature
35	**Phase 1: Scan and check inodes and block pointers**
9	*Bad block:* Check fields of bad-block inode; ensure superblocks and group descriptors in healthy blocks
18	*Inode structure:* Check fields (*e.g.*, mode, time, size) of different inodes (*e.g.*, root, reserved, boot load)
1	*Range check:* Ensure direct and indirect pointers point within the file system
7	*Conflicts:* Ensure no conflict among block pointers (*e.g.*, two inodes should not share a block)
38	**Phase 2: Scan and check all directory entries**
16	*Directory:* Check each has '.' and '..' entry, '.' points to itself, does not have missing block, fields of dir inode consistent (*e.g.*, acl, fragment size)
9	*Dir Entry:* Check each entry has correct name length, each points to an in-range inode, record length valid, filename contains legal characters
5	*Pathnames:* Each entry points to used inode, does not point to self, does not point to inode in bad block, does not point to root, dir has only one parent
8	*Special inodes:* Check device inodes and symlinks
6	**Phase 3: Ensure all directories are connected to the file system tree**
3	*lost+found*: Ensure lost+found directory is valid and ready to be populated
3	*Reattach*: Reattach orphan directory to lost+found
3	**Phase 4: Fix reference counts and reattach zero-linked file to lost+found**
11	**Phase 5: Check block and inode bitmaps against on-disk bitmaps**
121	**Total**

Table 3: **Repairs performed by fsck and SQCK$_{fsck}$.** *The table summarizes the 121 repairs performed by the traditional fsck.*

We have constructed an *information dependency graph* for the data structures in ext2 to ensure that repairs are performed in the correct order. A portion of this graph is shown in Figure 10. The figure illustrates that e2fsck does not follow the order specified by the dependency graph. SQCK$_{correct}$ reorders the relevant queries to ensure that single, double, and triple indirect blocks are all validated in the correct order before repairing the direct pointers themselves. We find that reordering repairs in SQCK is straightforward due to the structure of the queries; we do not believe such reordering is simple in e2fsck.

Currently, the dependency graph must be manually constructed by the file system developer/administrator. Since the repair queries in SQCK are neatly structured,

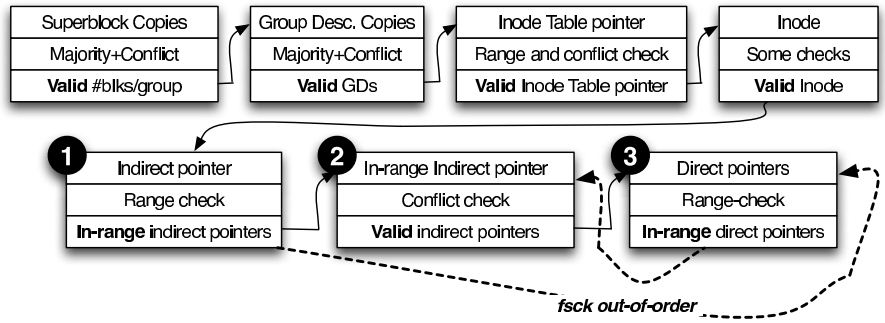


Figure 10: **Information dependency graph.** *The figure shows a chain of information dependency. Note that the full graph forms a tree-like graph; to save space, only a partial dependency chain is shown. Each box contains three rows: a new information obtained from the previous box, the check (and the corresponding repair, not shown), and the new state of the information after the check. For example, in box 1, an indirect pointer is obtained from a valid inode. After the range-check, the indirect pointer is marked in-range, but not yet valid. After it passes the conflict-check in box 2, it is finally marked as valid, which implies that we can proceed to box 3 which repairs out-of-range direct blocks contained in this indirect block. Unfortunately, e2fsck does not follow this ordering, as shown by the dashed lines; fsck proceeds repairing the direct pointers from a not yet valid indirect block. When e2fsck later finds out that the indirect block is indeed invalid (e.g., conflicting with other file system metadata), the content of the metadata has been accidentally corrupted in box 3.*

New Repair	LOC (C)	LOC (SQL)
Majority rule on block bitmap pointers	40	22
Majority rule on inode bitmap pointers	40	22
Majority rule on inode table pointers	40	22
Finding false parents	13	14
Reconstructing missing directories (*)	47	20
Precedence cloning	23	19
Secure cloning (**)	41	8

Table 4: **New repairs.** *The table lists all the new repairs we introduce. (*) In addition to the number of lines reported here, this new rule heavily uses the primitives as in Figure 9. (**) The number of lines reported for this rule is the additional code to the original cloning repair.*

```
SELECT  F.*
FROM    DirEntryTable P, DirEntryTable C,
        DirEntryTable F
WHERE   // P says C is his child
        P.entry_num >= 3      AND
        P.entry_ino  = C.ino  AND
        // and C says P is his parent
        C.entry_num  = 2      AND
        C.entry_ino  = P.ino  AND
        // but F, the false parent, says
        // C is also his child. P wins.
        F.ino <> P.ino        AND
        F.entry_num >= 3      AND
        F.entry_ino  = C.ino
```

Figure 11: **Finding false parents.** *This query returns the actual false parents. A false parent is a parent that claims to own a child even though the child is already strongly connected to another parent.*

the ordering can then be manually verified against the dependency graph. More ideally, a static tool could be built on top of SQCK to verify the ordering automatically. Specifically, each query could be tagged with a unique name that describes the check/repair performed, then a parser could automatically construct the ordering from the code, and finally a verifier could compare the constructed ordering against the specified ordering. This highlights that a structured fsck can be easily verified than a cluttered one.

3.3.3 Improving e2fsck

Our final version, SQCK$_{improved}$, improves how the file system is checked by utilizing more of the information that resides within the file system image. Table 4 lists the new information-complete, policy-consistent, and secure repairs in SQCK$_{improved}$.

The first three repairs utilize the replicas that ext2 keeps of the group descriptor blocks on disk. While e2fsck does examine these replicas if the primary copy is obviously corrupted, e2fsck misses opportunities to use correct replicas when the primary "looks" fine. Thus, SQCK$_{improved}$ always examines all replicas and performs majority voting across them to determine the correct values; this voting is performed for three important fields: the pointer to the data block bitmap, the inode bitmap, and the inode table. With these fixes, SQCK$_{improved}$ performs information-complete repairs when the pointer to the inode table is corrupted, as desired (described in Section 2). These new repairs are straightforward to implement, requiring only 22 lines of SQL and 40 lines of C.

The fourth repair utilizes the extra information kept in directory ".." fields to repair corrupted directories. First, we fix the false parenthood problem exhibited by e2fsck. With SQCK, we replace the incorrect check of e2fsck originally shown in Figure 7 with the one in Figure 11. This new query elegantly expresses relatively complex

```
SELECT  X.ino, X.start, X.end,
        V.start, V.end,
        (ABS(X.pBlk-V.start)) as distance
FROM    ExtentTable C,
        (A query that returns the start and
         end of a shared extent) AS V
WHERE   X.start <= V.start AND V.end <= X.end
ORDER BY V.start, distance
```

Figure 12: **Locality-aware repair.** *The query above returns shared blocks that are sorted based on the locality distance from the pointers. The inner query (not shown), stored in Table V, returns a the list of duplicate blocks. The* ABS *command helps sorting the result based on locality distance.*

behavior: it only returns false directory entries in which the child directory does not claim them as a parent with ".."; thus, this false directory entry is correctly cleared instead of that of the rightful parent.

We can extend this repair slightly to write the fifth repair, which corrects even more complicated corruptions of the directory hierarchy. For example, if a path `/a/b/c/` exists and `b`'s inode is corrupted such that `b` no longer appears to be a directory, e2fsck does not do any reconstruction and simply moves `c` to lost+found. However, $\text{SQCK}_{improved}$ completely reconstructs the contents of `b` from the back pointers of its children. The complete rule requires a total of 20 new SQL lines with C code similar to that shown in Figure 9.

The sixth repair corrects the allocation policy of e2fsck. Specifically, e2fsck clones data blocks without checking which file is closer to the shared data block. Ideally, the repair should give the existing block to the closest inode and allocate the new clone to the other inode. With SQCK, locality optimizations are easily performed. For example, Figure 12 shows how we utilize the ABS and ORDER BY SQL commands to calculate the distance between a block and its pointer. The bold text shows that the results are sorted on the start of the shared extents and then on the distance between the shared extent and the blocks that point to the extent (X.pBlk). Given this list, SQCK can easily perform the repair such that the shared extent is kept with its closest pointer.

Finally, the seventh repair adds secure cloning. This is done in two ways. First, suppose a corrupt direct pointer incorrectly points to a bitmap block; since the bitmap block is pointed to by more than one group descriptor replica, it is more likely the direct pointer is mistaken than all of the group descriptor replicas; therefore, cloning of that block simply leaks information and does not need to be performed.

Second, suppose a data block is shared by two inodes, one in the `/root` directory and one in the `/home/UserA` directory. In this case, if we want to prevent leaking of information, we might not want to clone the shared block,

```
first_block =  sb->s_first_data_block;
last_block = first_block + blocks_per_group;

for (i = 0, gd=fs->group_desc;
     i < fs->group_desc_count;
     i++, gd++) {
  if (i == fs->group_desc_count - 1)
    last_block = sb->s_blocks_count;
  if ((gd->bg_block_bitmap < first_block) ||
      (gd->bg_block_bitmap >= last_block)) {
      px.blk = gd->bg_block_bitmap;
      if (fix_problem(PR_0_BB_NOT_GROUP, ...))
        gd->bg_block_bitmap = 0;
  }
  ...
}
```

Figure 13: **C-version of Figure 3.** *The C fragment above shows the e2fsck's implementation of the "check block bitmap not in group" shown in Figure 3.*

```
if ((dot_state > 1) &&
    (ext2fs_test_inode_bitmap
     (ctx->inode_dir_map, dirent->inode))) {

  // ext2fs_get_dir_info is 20 lines long
  subdir = e2fsck_get_dir_info(dirent->inode);
  ...
  if (subdir->parent) {
    if (fix_problem(PR_2_LINK_DIR, ...)) {
      dirent->inode = 0;
      goto next;
    }
  } else {
    subdir->parent = ino;
  }
}
```

Figure 14: **C-version of Figure 7.** *The C fragment above shows the e2fsck's implementation of the "check multiple parents" shown in Figure 7.*

instead we remove the pointer from the user and keep the one from the root inode. In addition to the existing block conflict check and cloning primitives, this new rule only requires additional two SQL files, for a total of 8 lines to do the path traversal, and 41 lines of C code.

The secure clone repair could be seen as an example where an administrator's decision is more appropriate than an automated one. SQCK does not throw away the need to ask the administrator for the right decision. In such cases, different policies should be present for the administrator to choose from. In SQCK, we can execute different policies easily; each policy is simply mapped to a query or a set of queries.

3.4 Summary

In summary, we have found that declarative queries can succinctly express the many different types of checks and

Improving Scan Time	
1	Reduce seek time with sorted job queue
Improving Load Time	
2	Make the table content compact
3	Only load checked information
4	Use threads to exploit idle time
Improving Check Time	
5	Write queries that leverage indices
6	Leverage fs domain specific knowledge
7	Use bitmaps to reduce search space

Table 5: **Optimization Principles.** *The table lists the optimizations that we have performed such that performance of SQCK is competitive with e2fsck.*

repairs that fsck performs. Our experience also shows that writing checks and repairs in declarative queries is relatively straightforward; each query is written in a few iterative refinement. A complex check or repair, with a little bit of help from C code, can be broken into several short queries that are easy to understand. On average, each query we have written is 7 lines long, and the longest one is 22 lines. Furthermore, only 24 repairs require help from C code. The functionalities of the corresponding C code are generally simple; C code is only used to run a set of queries and iterate the query results. Note that this is different than how C code is used for cross-checking in e2fsck, as illustrated in Figure 13 and 14. Both of the code segments illustrate that a low-level C implementation tends to make a simple check hard to understand and debug. Given such examples, adding the new repairs described in the previous section into e2fsck is likely to complicate the code more.

4 Implementation

We now describe our implementation of the SQCK phases for scanning the file system image from disk, loading the database tables, checking and repairing the structures, and finally flushing the repairs to disk. Our current implementation of SQCK runs on top of a MySQL database and targets the ext2 file system in Linux 2.6.12. When describing our implementation, we focus on the optimizations we found were necessary for achieving respectable performance; Table 5 summarizes these optimizations across the phases.

4.1 Scanning and Loading

In our current implementation, SQCK combines its scanning and loading phases. Conceptually, SQCK maintains a queue of the structures that must be read from disk, processed, and loaded into the tables. As structures are processed, SQCK follows their pointers to determine the next structures. For example, the queue is initialized from the primary superblock; after the superblock, the locations of the group descriptor copies are known; subsequently, the inode tables are processed, which leads to individual inodes and their data blocks.

SQCK implements a number of performance optimizations for scanning and loading. First, to reduce the scan time, SQCK sorts the requests in the queue based on their on-disk locations; sorting the requests minimizes disk head positioning time, especially for file systems that are fragmented. We note that although e2fsck performs a partial optimization of this sort (*i.e.*, directory blocks are sorted before read from the disk [10]), e2fsck is not able to perform the same optimization (*e.g.*, indirect blocks still have to be traversed logically) because it heavily intermixes scanning with checking [19]. SQCK is able to optimize scanning because reading from disk is completely decoupled from checking; hence, SQCK does not need to follow structures in a logical manner.

The primary reason we decouple scanning from checking is because we want to make the common case fast; if corruption is a rare case than our approach improves the overall fsck time. However, there is a trade-off: if corruption is huge, extra work is needed to invalidate the garbage loaded into the database. Our design is not limited to that only approach; if desired, SQCK can be redesigned by intermixing some phases of scanning and checking according to the structural logical hierarchy. For example, when loading and checking indirect blocks, triple indirect blocks will be loaded and repaired in the database, then only valid double indirect blocks will be loaded to the database, and so on.

Second, SQCK improves load time (and check time) by reducing the size of the initial database tables. Our initial implementation loaded the ext2 structures to match their on-disk format; specifically, SQCK loaded each on-disk pointer as a direct pointer. However, we found that this approach made checking even 100 GB file systems unattractive. Therefore, our next optimization modified the tables to instead use extents to represent pointers referring to contiguous blocks.

Third, SQCK reduces loading time by only loading allocated meta-data. Given that most file systems are half-full [5], a great deal of the inodes are not actually used. To reduce the size of the tables, SQCK does not load the unused inodes into the database tables (though it of course still scans them from disk). However, e2fsck performs one check on unused inodes that SQCK must be able to replicate: e2fsck verifies that each inode with a link count of zero also has a deletion time of zero. To handle this repair, SQCK performs this one check during processing. If SQCK finds a non-conforming inode, that inode is loaded into the table on the fly; to mark that the inode has been repaired, its *used* field is cleared and the

dirty field is set. We note that this optimization is consistent with the direction in which future file systems are going: ext4 explicitly marks unallocated sections of the inode table to help e2fsck run more efficiently [4].

Fourth, the scanner-loader in SQCK is multi-threaded. Each thread within the pool is able to independently grab a structure from the queue, read the data from disk, process it, and load the information into the corresponding table. Multiple threads allow SQCK to overlap reading requests from disk with loading the tables. As we will see in our evaluation, this optimization is especially important for large partitions.

4.2 Checker

After all metadata has been uploaded into the database tables, SQCK initiates the checking phase, which runs the queries as discussed in the previous section. One important note is that since the checker runs only after the loader, corrupt data can be loaded into the tables. Hence, SQCK provides primitives to invalidate a structure along with the information that originates from it. For example, if the block number that points to an inode table is corrupt, the wrong inodes and the wrong data pointers will be loaded into the table. Later, when the checker discovers that the inode table pointer is corrupt, it simply calls the SQCK primitives to invalidate the corresponding inodes, extents, and directory entries.

The checker has been optimized for performance in three main ways. First, we have found that SQCK must contain appropriate indices for each table; without indices a full scan must be done for each check and joining multiple tables requires a very long time. Thus, each table contains indices over the fields that are checked with the comparison operators.

Given the indices, some queries must be rewritten to leverage them. In our experience, MySQL is not able to always extract the implicit index comparisons in some queries. For example, the check that no directory entry points to an unused inode was originally written as shown in the top half of Figure 15. When the rule was rewritten to make the index comparison explicit, as shown in the bottom part of the figure, the query time improved significantly. Thus, making index comparison explicit is an important principle to do fast checking. We rewrote a total of four queries in this manner, reducing the check time for those four queries from 72 seconds down to just 0.09 seconds on a 1 GB partition.

Second, we have found it beneficial to incorporate file system domain knowledge into the queries. One example is the rule that counts how many blocks are being used in a group. Since SQCK uses extents, it must first select the extents in that group. The naive range-checking query could be written as follows: `(G.start <= X.start AND X.end <= G.end)`. However, given that we know valid extents cannot overlap group boundaries (this has been verified in previous queries), the range-check query can be simplified to `(G.start <= X.start <= G.end)`. This simplified query improves check performance.

```
// find an entryIno that is in the list of
// unused inodes
SELECT *
FROM   DirEntryTable
WHERE  entryIno IN
         (SELECT ino
          FROM   InodeTable
          WHERE  used = 0)
              ---- vs. ----
// find an entryIno that exists in the
// InodeTable and the used field is zero
SELECT *
FROM   DirEntryTable
WHERE  EXISTS
         (SELECT *
          FROM   InodeTable AS I
          WHERE  I.ino  = D.entryIno AND
                 I.used = 0)
```

Figure 15: **Explicit index comparison.** *We rewrite the code to unearth the index comparison.*

The final optimization addresses how to join tables where an index comparison is not possible. For example, the query finding shared blocks across files joins the ExtentTable with itself to find any overlapping extents. We optimize this query by making the search space smaller with bitmaps. For this example, SQCK uses two bitmaps: one for marking used blocks and one for marking shared blocks; the latter bitmap provides a hint as to which extents have overlapping blocks. To find out which part of an extent is actually overlapping, SQCK joins the resulting small table with itself.

4.3 Flusher

Finally, SQCK needs to update any repaired structures to the disk. SQCK is able to determine which structures have been modified by selecting those entries where the dirty flag is set. Following the same behavior as e2fsck, SQCK updates the structures in-place on disk (*i.e.*, it does not currently use a separate journal).

To ensure the metadata writes are ordered correctly [16], currently SQCK performs a series of queries ordered by the dependency graph; the graph ensures that blocks are updated before the pointers to those blocks. In the next generation of SQCK, a journaling facility will be added to ensure that a crashed repair process will not modify the old data partially.

	C Code		SQL
Component	**LOC**	**; count**	**LOC**
Scanner	2759	1378	–
Loader	609	177	103
Checker+Repair	*2527	1468	910
Primitives	695	348	98
Flusher	114	49	27
Total	6704	3420	1138

Table 6: **SQCK$_{improved}$ LOC.** *The table presents the complexity of SQCK$_{improved}$. Scanner includes threads and functions that process the structures. (*) The C code for the checkers and repairs are mostly wrappers that call the SQL files.*

	SQCK	ext2	ReiserFS	XFS
LOC	2527	16472	11281	21773
# Chks	121	121	156	344
Instr. gap	16 ± **16**	71 ± **161**	56 ± **203**	128 ± **257**
Func. gap	1 ± **1**	4 ± **6**	1 ± **3**	5 ± **6**
# Chk func.	121	31	32	72
# Chks/chk-func	1 ± **0.1**	4 ± **5**	5 ± **8**	5 ± **5**

Table 7: **Checkers complexity.** *The table shows the logical complexity of SQCK$_{correct}$, ext2, ReiserFS and XFS checker codes (excluding libraries). Standard deviation is shown right next to the ± sign. "Inst. and Func. gaps" quantify the number of C instructions and functions separating one check from the next check. "# Chk func" shows in how many functions the checks are diffused. Finally, "# Chks/func" averages the number of checks performed in each checker function.*

5 Evaluation

In this section we evaluate SQCK along three axes: complexity, robustness, and performance. Overall, we believe many of the goals of SQCK have been achieved.

5.1 Complexity

Table 6 presents the complexity of SQCK$_{improved}$, the most complete version of SQCK. As the table suggests, SQCK is comprised of C and SQL code. The scanner is the only place where the complexity of the C code still exists. However, the code is generally simple because it scans the file system in a logical hierarchy. The checker code looks big, however, it is mostly wrapper functions that call the corresponding queries; most wrappers consist of the same 15 lines of C code. A generic wrapper could be built to reduce the amount of C code.

SQCK so far has been written all at once by one small group. Thus, it is possible that SQCK will become more complex when developed by a bigger group over a longer period of time. However, we believe the core power of SQCK lies within the simple and robust queries; each query consists of 7 lines of code on average. These queries decouple the checks from the C code, enabling us to maintain reliability in an easier way. Compared to e2fsck, which consists of 16 thousand LOC of cluttered checks and repairs and 14 thousand LOC of scan utilities, all written in low-level C code, SQCK can be considered a big step towards simplifying file system checkers.

To show that we are solving a broader significant problem, Table 7 attempts to quantify the logical complexity of ext2, ReiserFS, and XFS checker utilities, all written in C. The metrics shown in the table are generated by our parser written using CIL [25]. In fsck-related code, we annotate the location where each check is performed. The parser computes the complexity-metrics as described in the table. For example, we compute how many instructions and function calls separate each neighboring checks. If the numbers are high, the checks are most likely diffused and reasoning about their correctness might be nontrivial, if not impossible. The numbers reported in Table 7 exclude fsck libraries (*e.g.*, scanner), hence they only depict the logical complexity of the checker component.

We make two important observations: First, the average number of C instructions and functions that separate two checks are high in all fsck utilities, with significant standard deviations; the separation can be as low as 4 or as high as 1700 instructions. Second, checks are greatly diffused in many functions; a function could make a small number of checks while some other could perform as many as 47 checks. In such implementations verifying that all checks are complete and ordered correctly can be cumbersome. On the other hand, SQCK hides the complex logic of the checks in declarative queries, greatly reducing the gap between neighboring checks; the standard deviations shown in the SQCK column illustrate the neat organization we have achieved. In summary, we believe all C-implementations of fsck are likely to suffer from the same problems as e2fsck.

5.2 Robustness

To test the robustness of the three versions of SQCK, we have injected a total of 356 corruption scenarios. In each scenario, we injected corruption into one or more fields to test whether a check and the corresponding repair behave as expected. First, we have injected 55 faults to ensure that SQCK$_{fsck}$ passes most of the interesting repairs out of the total 121 repairs. Injecting faults for the rest of the tests should be straightforward. Second, we have verified that SQCK$_{correct}$ passes the fsck reliability benchmark described in Section 2 by injecting additional 180 faults. Finally, we injected 10 new faults to test the new repairs that we introduced in SQCK$_{improved}$ and verified the resulting repairs.

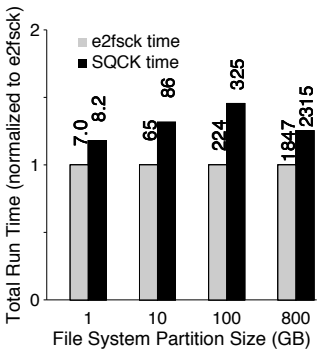


Figure 16: **Overall runtime comparison .** *The bar graph shows the comparison of the total runtime of e2fsck and fully optimized SQCK for different file system sizes. The bars are all normalized to e2fsck runtime and the SQCK bar show the relative slowdown. The absolute runtime figures in seconds are shown on top of the bars.*

5.3 Performance

The experiments in this section were performed on an 2.2 GHz AMD Opteron machine with 1 GB memory and 1 TB WDC WD10EACS disk. We used Linux 2.6.12, e2fsck 1.39, and MySQL 5.0.51a. The tables are mounted on a 512 MB ramdisk.

We test the performance of SQCK and e2fsck on four partitions with different sizes: 1, 10, 100, and 800 GB. Each of the partition is made half-full [5] by filling it with the root file system image of a machine in our laboratory along with a large number small files from kernel builds and large files from virtual machine images.

Figure 16 shows e2fsck compared to our fully optimized SQCK. The fully optimized SQCK incorporates all the principles described in Table 5; specifically, it sorts the block scan, loads extents and linked inodes only, uses 16 worker threads, and uses fast queries. In our first generation prototype we managed to keep the running time of SQCK within 1.5 times of e2fsck runtime.

We show in more detail how each of the scan and load optimization principles improve the runtime significantly by turning off one optimization feature at a time. The runtime of each of these unoptimized versions are compared relative to the fully optimized SQCK.

First, the sorted job queue is disabled such that we scan the file system logically. Figure 17 shows that for a large file system (*e.g.*, 800 GB), sorting the job queue plays a significant role; scanning the file system logically takes almost 3 times as long as the fully optimized one. Note that in this experiment, we disabled the loading phase to compare only the scan performance. The serial scanning for the 100 GB file system is 8 seconds faster than the fully optimized SQCK because the file system was almost not fragmented at all; the advantage of the sorted scanning is noticeable for fragmented and/or big file systems.

Second, we show the importance of making the initial table compact. Figure 18 shows the slowdown of two unoptimized versions: one that loads all inodes, and one that loads direct pointers instead of extents. When loading all inodes, the runtime is increased significantly; for 800 GB file systems, 97 million inodes will be loaded out of which only 900 thousand have non-zero link counts. When loading direct pointers, the runtime increases dramatically. For the 100 GB file system, the DirectPointerTable already consumes 360 MB, while the ExtentTable only consumes 9 MB.

Third, Figure 19 shows how multiple threads enable us to significantly overlap scan and load time. When the number of worker threads is reduced to one, the slowdown is almost 1.5 times in all file systems. For large file systems, increasing the number of threads gives a faster runtime; at 800 GB, using 16 worker threads improves the runtime.

In summary, our evaluation of the first generation prototype of SQCK shows that SQCK obtains comparable performance to e2fsck. In the next generation of SQCK, we plan to perform two additional enhancements. First, some checks can be merged so that the table-scan time can be reduced. If the checks find a problem, then nested sub-checks will be run to pinpoint the actual problem. Second, we plan to run some checks and repairs concurrently by utilizing the information dependency graph in Figure 10. The graph provides the dependency tree that tells which checks and repairs are safe to run in parallel. With a faster overall check time, we hope file system developers will be encouraged to write as many rules as needed.

6 Related Work

We believe that SQCK is the first tool to apply a declarative query language for checking and repairing file systems. We briefly compare SQCK to research on specification and declarative languages and efforts focused on improving fsck.

Specification languages like Alloy [21] and Z [11] are useful for describing constraints of a system and then finding violations of that model. Similar in goals to SQCK, Demsky and Rinard took this approach for automatically repairing file systems [12]; however, we believe the drawback of their work is that it does not allow one to naturally express the *repairs* that should be performed when violations are discovered.

Numerous researchers have recently explored the advantages of using declarative languages in other domains, such as system configuration [13] and network overlays [23].

Finally, other researchers have proposed a technique for optimizing the performance of file system check-

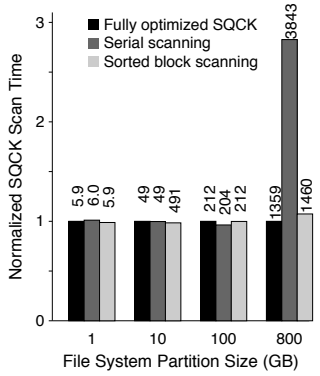


Figure 17: **Scan time improvement with sorted queue.** *This bar graph shows the time to scan each file system without loading. In each set, the left-most to right-most bars show the fully optimized SQCK, the logical scan, and the sorted scan with only 1 thread. The values are normalized to the fully optimized SQCK time.*

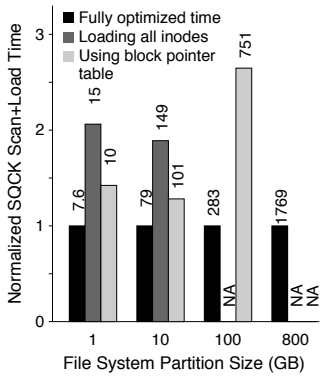


Figure 18: **Making the table compact.** *The bar graph shows the slowdown of scan and load time when we load big tables. In each set, the left-most to right-most bars show the fully optimized SQCK, fully optimized SQCK but with loading all inodes, and loading direct pointers instead of extents. The values are normalized to the fully optimized SQCK time. "NAs" imply experiments that do not finish in 3 hours.*

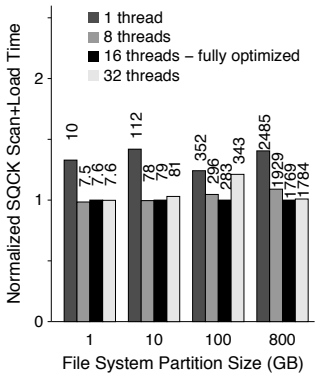


Figure 19: **Overlapping scan and load time.** *The bar graphs show the runtime of different runs that use different number of threads. With one thread, the runtime is the worst as we cannot utilize the idle time during scanning. The values shown are normalized to fully optimized SQCK time with 16 threads.*

ers [20, 26]. The basic approach in both checkers is to track which portions of the disk are being written to and focus repair on only those active portions of the disk. While this technique can certainly improve performance, it does not simplify the implementation of the checker nor enable fundamentally new repairs, as does SQCK.

7 Conclusion

Recovery code is complex and hard to get right. Current approaches describe recovery at a very low-level: thousands of lines of C code. One approach to improving the state of the art is to apply more formal techniques that find bugs in such code [14] and thus evolve the code towards a less-buggy future.

We instead advocate a higher-level strategy. By encapsulating the logic of a file system checker in a set of declarative queries, we provide a more concise description of what the checker should do. Complexity is the enemy of reliability; by paring down the checker to its declarative core, we believe we have taken an important step towards improving the robustness of file system checking.

Acknowledgments

We thank the anonymous reviewers and Greg Ganger (our shepherd) for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We also thank the members of the ADSL research group, in particular Nitin Agrawal, Sriram Subramanian, and Swaminathan Sundararaman for their insightful comments. Finally, we thank Yupu Zhang who helped us in annotating the XFS fsck.

This material is based upon work supported by the National Science Foundation under the following grants: CCF-0621487, CNS-0509474, CCR-0133456, as well as by generous donations from Network Appliance and Sun Microsystems.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] http://www.mysql.com/.

[2] http://e2fsprogs.sourceforge.net/.

[3] http://en.wikipedia.org/wiki/SQL.

[4] http://en.wikipedia.org/wiki/Ext4.

[5] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A Five-Year Study of File-System Metadata. In *FAST '07*, San Jose, CA, February 2007.

[6] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *SIGMETRICS '07*, San Diego, CA, June 2007.

[7] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and

Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *FAST '08*, pages 223–238, San Jose, California, February 2008.

[8] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Systematically Benchmarking the Effects of Disk Pointer Corruption. In *DSN '08*, Anchorage, Alaska, June 2008.

[9] George Candea and Armando Fox. Crash-Only Software. In *HotOS IX*, Lihue, Hawaii, May 2003.

[10] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In Proceedings of the First Dutch International Symposium on Linux, 1994.

[11] Jim Davies and Jim Woodcock. *Using Z: Specification, Refinement, and Proof* . Prentice Hall, 1996.

[12] Brian Demsky and Martin Rinard. Automatic Detection and Repair of Errors in Data Structures. In *OOPSLA '03*, Anaheim, California, October 2003.

[13] John DeTreville. Making system configuration more declarative. In *HotOS X*, Sante Fe, New Mexico, June 2005.

[14] Dawson Engler and Madanlal Musuvathi. Static Analysis versus Software Model Checking for Bug Finding. In *VMCAI '04*, Venice, Italy, January 2004.

[15] Rob Funk. fsck / xfs. http://lwn.net/Articles/226851/.

[16] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *OSDI '94*, pages 49–60, Monterey, CA, November 1994.

[17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP '03*, pages 29–43, Bolton Landing, NY, October 2003.

[18] Roedy Green. EIDE Controller Flaws Version 24. http://mindprod.com/jgloss/eideflaw.html, February 2005.

[19] Val Henson. The Many Faces of fsck. http://lwn.net/Articles/248180/, September 2007.

[20] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *HotDep II*, Seattle, Washington, Nov 2006.

[21] Daniel Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

[22] Kimberly Keeton and John Wilkes. Automating data dependability. In *Proceedings of the 10th ACM-SIGOPS European Workshop*, pages 93–100, Saint-Emilion, France, September 2002.

[23] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing Declarative Overlays. In *SOSP '05*, Brighton, UK, October 2005.

[24] Marshall Kirk McKusick, Willian N. Joy, Samuel J. Leffler, and Robert S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[25] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. Cil: An infrastructure for c program analysis and transformation. In *CC '02*, pages 213–228, April 2002.

[26] J. Kent Peacock, Ashvin Kamaraju, and Sanjay Agrawal. Fast Consistency Checking for the Solaris File System. In *USENIX '98*, pages 77–89, New Orleans, LA, June 1998.

[27] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *SOSP '05*, pages 206–220, Brighton, UK, October 2005.

[28] American Data Recovery. Data loss statistics. http://www.californiadatarecovery.com/content/adr_loss_stat.html.

[29] David M. Smith. The Cost of Lost Data. http://gbr.pepperdine.edu/033/dataloss.html.

[30] Nisha Talagala and David Patterson. An Analysis of Error Behaviour in a Large Storage System. In *The IEEE Workshop on Fault Tolerance in Parallel and Distributed Systems*, San Juan, Puerto Rico, April 1999.

[31] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *OSDI '06*, Seattle, WA, November 2006.

[32] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI '04*, San Francisco, CA, December 2004.

Transactional Flash

Vijayan Prabhakaran, Thomas L. Rodeheffer, Lidong Zhou
Microsoft Research, Silicon Valley
{*vijayanp, tomr, lidongz*}*@microsoft.com*

Abstract

Transactional flash (TxFlash) is a novel solid-state drive (SSD) that uses flash memory and exports a transactional interface (`WriteAtomic`*) to the higher-level software. The copy-on-write nature of the flash translation layer and the fast random access makes flash memory the right medium to support such an interface. We further develop a novel commit protocol called* cyclic commit *for TxFlash; the protocol has been specified formally and model checked.*

Our evaluation, both on a simulator and an emulator on top of a real SSD, shows that TxFlash does not increase the flash firmware complexity significantly and provides transactional features with very small overheads (less than 1%), thereby making file systems easier to build. It further shows that the new cyclic commit protocol significantly outperforms traditional commit for small transactions (95% improvement in transaction throughput) and completely eliminates the space overhead due to commit records.

1 Introduction

Recent advances in NAND-based flash memory have made solid state drives (SSDs) an attractive alternative to hard disks. It is natural to have such SSDs export the same block-level read and write APIs as hard disks do, especially for compatibility with current systems. SSDs are thus simply "hard disks" with different performance characteristics.

By providing the same API as disks, there is a lost opportunity of new abstractions that better match the nature of the new medium as well as the need from applications such as file systems and database systems. In this paper, we propose a new device called Transactional Flash (TxFlash) that exports such an abstraction. TxFlash is an SSD exposing a linear array of pages to support not only read and write operations, but also a simple *transactional* construct, where each transaction consists of a series of write operations. TxFlash ensures atomicity, *i.e.*, either all or none of the write operations in a transaction are executed, and provides isolation among concurrent transactions. When committed, the data written by a transaction is made durable on the SSD.

The atomicity property offered by the transactional construct has proven useful in building file systems and database systems that maintain consistency across crashes and reboots. For example, a file creation involves multiple write operations to update the metadata of the parent directory and the new file. Often, a higher-level system employs copy-on-write (CoW) or a variant of it, such as write-ahead-logging (WAL) [13], to ensure consistency. The essence of these mechanisms is to avoid *in-place* modification of data.

Although known for decades, these mechanisms remain a significant source of bugs that can lead to inconsistent data in the presence of failures [30, 16, 29], not to mention the redundant work needed for each system to implement such a mechanism. For example, in Linux, the common journaling module (jbd) is used only by Ext3 [28], although there are several other journaling file systems, such as IBM JFS [3] and XFS [26].

Having a storage layer provide a transactional API reduces the complexity of the higher level systems significantly and improves the overall reliability. Indeed, previous work has attempted to provide such constructs on hard disks using CoW and logging [5, 6, 22]. Unfortunately, one common side effect of CoW techniques is the *fragmentation* of the linear address space, *i.e.*, CoW tends to scatter related pieces of information over the disk when updating them. Reading those fragmented pieces of related information requires seeks, leading to poor performance. To mitigate this performance problem, systems that implement CoW also employ some form of checkpointing and cleaning [20], where related pages are reorganized into their home locations. However, cleaning costs can themselves be high [24].

The significant extra complexity at the disk controller layer and the poor read performance due to fragmentation are some of the main obstacles for providing a transactional API on hard disks. SSDs mitigate both problems, making it an ideal medium for supporting transactions. Modern SSD controllers already implement variants of CoW for performance reasons. Furthermore, fragmentation does not lead to performance degradation on SSDs because random read accesses are fast.

We develop a new *cyclic commit* protocol for TxFlash to allow efficient implementation of transactions. Cyclic commit uses per-page metadata to eliminate the need for a separate commit record as in a standard commit protocol. Traditionally, the existence of a commit record is used to judge whether a transaction is committed or not. In our cyclic commit protocol, the judgment is based on the metadata stored along with the data. The result is a better commit protocol in terms of performance and space overheads.

We evaluate TxFlash in three complementary ways.

- We formally specify the cyclic commit protocol in TLA+ and check it using the TLC model checker [10].
- We design and implement TxFlash as an extension to an SSD simulator from previous work [1]. The resulting simulator is used to compare the variants of cyclic commit with traditional commit, as well as to assess the overheads of transactional support by comparing TxFlash with the basic SSD.
- We develop TxExt3, a version of the Linux Ext3 file system modified to exploit TxFlash's transactional API. To obtain realistic end-to-end performance numbers, we run TxExt3 on a real SSD using an intermediate pseudo-device driver to emulate the TxFlash firmware.

Compared to traditional commit, cyclic commit improves the performance significantly, especially for small transactions, while eliminating the space overhead completely. For transactions less than 100 KB in size, cyclic commit improves transaction throughput by 95% over traditional commit. For transactions larger than 4 MB, cyclic commit performs as well as traditional commit. Our simulation results show that TxFlash can provide the transactional capabilities with negligible overhead (less than 1%) when compared to an SSD. Finally, for I/O intensive synchronous applications, TxExt3 reduces the run time by as much as 65%, in both data and metadata journaling modes. However, the benefits for compute-intensive applications are small (about 2%).

The rest of the paper is organized as follows. Next, we make a case for TxFlash and explain its API and architecture (§2). The core cyclic commit protocol and its variations are the subject of the following section (§3). We follow this with the details of the implementation (§4) and evaluation (§5). Then, we cover the related work (§6) and finally conclude (§7).

2 The Case for TxFlash

The rationale for TxFlash is deeply tied to the fundamentals of the flash based SSDs, which are covered in this section. We then describe the API and the architecture of TxFlash, followed by a discussion of the rationale for TxFlash. For the rest of the paper, we interchangeably use "page" or "logical page" to refer to pages as used by a higher-level system. A page on the stable storage is referred as a "physical page" or "flash page".

2.1 SSDs: A Primer

Similar to disks, NAND-flash based SSDs provide a persistent medium to store data. Unlike disks, however, SSDs have no mechanically moving parts, yielding drastically different performance characteristics from disks.

An SSD consists of multiple flash packages that are connected to a controller, which uses some volatile memory for buffering I/O requests and maintaining internal data structures. A flash package is internally made up of several planes, each containing thousands of blocks; each block in turn consists of many 4 KB pages. In addition to the data portion, a flash page also contains a metadata portion: for every 4 KB page, there is a corresponding 128 bytes for metadata such as checksums and error correction codes. Reads and writes can be issued at page granularity and SSDs can be designed to ensure atomicity for a single page write, covering both the data and the metadata.

Another characteristic of NAND-flash is that, after a physical page has been written, it must be *erased* before any subsequent writes, and erasures must be performed at the block granularity. Since a block erase is costly (1.5 ms), SSDs implement a *flash translation layer* (FTL) that maintains an in-memory *remap table*, which maps logical pages to physical ones. When a logical page is written, the FTL writes the data to a new physical page and updates the mapping. Essentially, the FTL implements CoW to provide the illusion of in-place writes and hide the cost of block erasures.

The in-memory remap table must be reconstructed during boot time. An SSD can use the metadata portion of a physical page to store the identity and version number of the logical page that is stored.

The FTL further maintains a list of free blocks. Because of CoW, obsolete versions of logical pages may be present and should eventually be released to generate free space. SSDs implement a *garbage collection* routine that selects a block, copies valid pages out of the block, erases it, and adds it to the free-blocks queue. The remapping and garbage collection techniques are also used to balance the wearing down of different blocks on the flash, often referred to as *wear-leveling*.

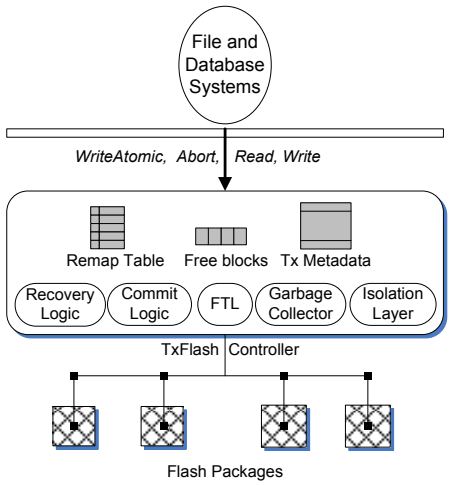


Figure 1: **Architecture of a TxFlash Device.** *The controller runs the logic for ensuring atomicity, isolation, and recovery of transactions. Data and metadata are stored on the flash packages.*

2.2 TxFlash API and Architecture

Figure 1 shows a schematic of TxFlash. Similar to an SSD, TxFlash is constructed with commodity flash packages. TxFlash differs from an SSD only in the API it provides and in the firmware changes to support the new API.

Transactional Model. TxFlash exports a new interface, `WriteAtomic`$(p_1 \ldots p_n)$, which allows an application to specify a transaction with a set of page writes, p_1 to p_n. TxFlash ensures atomicity, *i.e.*, either all the pages are written or none are modified. TxFlash further provides isolation among multiple `WriteAtomic` calls. Before it is committed, a `WriteAtomic` operation can be aborted by calling an `Abort`. By ensuring atomicity, isolation, and durability, TxFlash guarantees consistency for transactions with `WriteAtomic` calls.

In addition to the remap table and free-blocks queue maintained in an SSD, TxFlash further keeps track of in-progress transactions. When a transaction is in progress, the isolation layer ensures that no conflicting writes (*i.e.*, those updating the same pages as the pending transactions) are issued. Once a transaction is committed, the remap table is updated for all the pages in the transaction.

At the core of TxFlash is a commit protocol that ensures atomicity of transactions despite system failures. A commit protocol includes both a commit logic that is executed when a transaction is committed, and a recovery logic that is executed at boot time. The latter reconstructs the correct mapping information based on the information persisted on the flash. The actions of other modules, such as the garbage collector, depend on the actual commit protocol used.

2.3 Rationale for TxFlash

There are two main points in favor of TxFlash: utility and efficiency. TxFlash is useful because its API can benefit a large number of storage applications. TxFlash is efficient because the underlying SSD architecture matches the API well.

Interface Design. We choose to support a limited notion of transactions in TxFlash because we believe this choice reflects a reasonable trade-off between complexity and usefulness. The `WriteAtomic` interface is desirable to any storage system that must ensure consistency of multi-page writes despite system failures; file systems and database systems are known examples.

We choose not to implement full-fledged transactions, where each transaction consists of not only write operations, but also read operations. This is because they introduce significant additional complexity and are overkill for applications such as file systems.

Compatibility is often a concern for a new API. This is not an issue in this case because we preserve the simple disk APIs so that existing systems can be run directly on TxFlash. However, by using the additional transactional constructs certain parts of a system can be made simpler and more efficient. We show later in the paper (§4) that while Ext3 can be run directly on TxFlash, parts of its journaling code can be simplified to use the new `WriteAtomic` construct.

Transactions on SSDs. Compared to hard disks, SSDs are particularly ideal for supporting transactions for the following reasons.

• Copy-on-write nature of SSDs. Extending a log-structured system to support transactions is not new [23]. The FTL already follows the CoW principle because of the write-erase-write nature of the flash pages and wear-leveling. Extending FTL to support transactions introduces relatively little extra complexity or overhead.

• Fast random reads. Unlike hard disks, fragmentation is not an issue in SSDs, again because of their inherent solid-state nature: an SSD can rapidly access random flash-memory locations in constant time. Although SSDs perform cleaning for freeing more re-usable space, there is no need for data re-organization for locality.

• High concurrency. SSDs provide a high degree of concurrency with multiple flash packages and several planes per package, and multiple planes can operate concurrently. Enabled with such high parallelism, SSDs can support cleaning and wear-leveling without affecting the foreground I/O.

• New interface specifications. Traditional storage interfaces such as SATA do not allow the devices to export new abstractions. Since SSDs are relatively new, alternative specifications can be proposed, which may provide the freedom to offer new device abstractions.

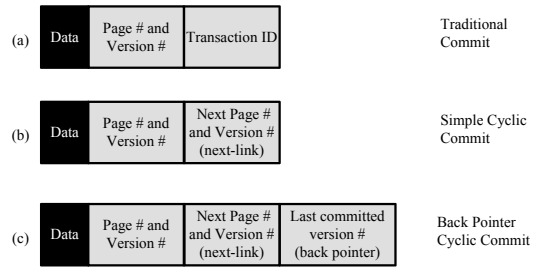


Figure 2: **Page Format.** *Information stored in the metadata portion of a flash page by various commit protocols.*

3 Commit Protocols

To ensure the atomicity of transactions, TxFlash uses a commit protocol. The protocol specifies the steps needed to commit a transaction, as well as a recovery procedure. The recovery procedure is executed after a system reboot to determine which transactions are committed based on the persistent state on the storage. Commit protocols tend not to update the data in-place and therefore invariably require a separate garbage collection process, whose purpose is to release the space used by obsolete or aborted transactions.

3.1 Traditional Commit (TC)

Modern journaling file systems such as Ext3 [28], IBM JFS [3], XFS [26], and NTFS [25] use a form of redo logging [13] for atomicity; we refer to this mechanism as the *traditional commit* protocol. In traditional commit, new data for each page is written to the storage device as an *intention record*, which contains a data portion and a metadata portion. The metadata portion stores the identity of the page and the transaction ID as shown in Figure 2(a). Once all the writes of the intention records have completed successfully, a *commit record* is written to the storage device. Once the commit record is made persistent, the transaction is committed. When recovering during a reboot, traditional commit decides whether a transaction is committed or not based on the existence of the corresponding commit record.

Typically, the intention records and the commit records are written into a log. The updates in committed intention records are written in-place to the home locations by a *checkpoint* process. Once checkpointing is completed, all the intention records and commit records are garbage collected by truncating the log.

Traditional Commit on SSDs. With an SSD as the underlying storage device, thanks to the indirection provided by the remap table, no separate checkpointing is necessary: the logical pages can be remapped to the new locations when a transaction commits. Also, all writes within the same transaction can be issued concurrently, thereby exploiting the inherent parallelism on an SSD.

However, the need for the separate commit record in traditional commit may become particularly undesirable. The commit record write must be issued only after all the intention record writes are completed; such write ordering introduces the latency of an extra write per transaction. Because of the absence of a separate checkpointing process, a special garbage collection process is needed for commit records: a commit record can be released only after all the intention records of the corresponding transaction are made obsolete by later transactions. Both the performance and space overheads introduced by commit records are particularly significant for small transactions. Group commit [7] was designed to reduce some of these overheads but it works well only when there are multiple operations that can be delayed and grouped together.

3.2 Simple Cyclic Commit (SCC)

Instead of using commit records to determine whether a transaction is committed or not, a *cyclic commit* protocol stores a link to the next record in the metadata of an intention record (*i.e.*, the logical page of an SSD) and creates a cycle among the intention records of the same transaction. This eliminates the need for a separate commit record for each transaction, thereby removing the space and performance overheads.

Figure 2(b) shows the intention record used by the cyclic commit, where the next page and version numbers are additionally stored in the metadata portion as the *next-link*. For each transaction, the next-link information is added to the intention records before they are concurrently written. The transaction is *committed* once all the intention records have been written. Starting with any intention record, a cycle that contains all the intentions in the transaction can be found by following the next-links. Alternatively, the transaction can be *aborted* by stopping its progress before it commits. Any intention record belonging to an aborted transaction is *uncommitted*.

In the event of a system failure, TxFlash must be restarted to recover the last committed version for each page. The recovery procedure starts by scanning the physical pages and then runs a recovery algorithm to classify the intention records as committed or uncommitted and identify the last committed version for each page based on the metadata stored in the physical pages.

We use $\mathcal{S}$ to refer to the set of intention records (in terms of their logical page numbers and versions) obtained by scanning the stable storage and $\mathcal{R}$ for the set of intention records that are referenced by the next-link field of any intention record in $\mathcal{S}$. All records in $\mathcal{S} \ominus \mathcal{R}$ are present on the storage, but not referenced by any other record ($\ominus$ represents set difference); similarly, all records

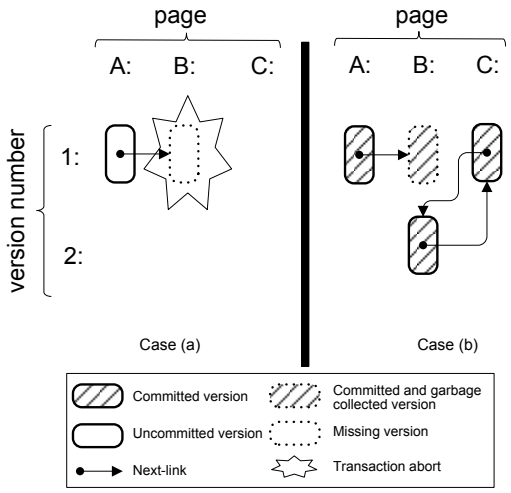


Figure 3: **Ambiguous Scenario of Aborted and Committed Transactions.** *Two cases with a broken link to B_1: an aborted transaction on the left, where B_1 was not written; a committed transaction on the right, where B_1 was superseded and cleaned.*

in $\mathcal{R} \ominus \mathcal{S}$ are referenced by some record in the stable storage but not present themselves. For an intention record r, we use $r.next$ to refer to the next-link in r.

The following Cycle Property from cyclic commit is fundamental to the recovery algorithms. It states that the classification of an intention record can be inferred from its next-link; this is because they belong to the same transaction. It further states that, if a set of intention records forms a cycle, then all of them are committed.

Cycle Property. For any intention record $r \in \mathcal{S}$, r is committed if and only if $r.next$ is committed.

If there exists a set T of intention records $\{r_i \in \mathcal{S} \mid 0 \leq i \leq n-1\}$, such that for each $0 \leq i \leq n-1$, condition $r_i.next = r_{(i+1) mod\ n}$ holds, then any $r \in T$ is committed.

It is worth noting that a break in a cycle (i.e., $r \in \mathcal{S}$ and $r.next \notin \mathcal{S}$) is not necessarily an indication that the involved intention record is uncommitted. Figure 3 illustrates this case. In this example, pages are referred to by the letters A through C and the version numbers are 1 and 2. Next links of intention records are shown by arrows. In this Figure, various versions are labeled as to whether they are committed (crosshatch fill) or uncommitted (white fill). Missing versions are indicated by a dotted border. We use "P_i" for version i of page P. Consider the scenario where A_1 has its next-link set to B_1, but B_1 does not exist on the SSD. There are two cases that could lead to the same ambiguous scenario: in the first case, as shown in Figure 3(a), the transaction with A_1 and B_1 was aborted and B_1 was never written; in the second case, as shown in Figure 3(b), the transaction with A_1 and B_1 commits, followed by another successful transaction that creates B_2 and C_1, making B_1 obsolete and causing B_1 to be garbage collected. In the first case, A_1 belongs to an aborted transaction and should be discarded, while in the second case A_1 belongs to a committed transaction and should be preserved.

Observe that an intention record P_i can be garbage collected only when there is a higher version P_j ($j > i$) that is committed. SCC is based on the following SCC Invariant, which is ensured by correct initialization and handling of uncommitted intention records.

SCC Invariant: If $P_j \in \mathcal{S}$, any intention record $P_i \in \mathcal{S} \cup \mathcal{R}$ with $i < j$ is committed.

SCC Initialization. When TxFlash starts for the first time, it initializes the metadata of each page by setting the version number to 0 and the next-link to itself.

Handling Uncommitted Intention Records. If an intention record $P_i \in \mathcal{S} \cup \mathcal{R}$ belongs to an aborted transaction, to preserve the SCC Invariant, before a newer version of P is written, P_i and Q_j must be erased, where $Q_j.next = P_i$. This avoids misclassification of P_i (due to the newer version of P) and Q_j (by following the next link). That is, any uncommitted intention on the stable storage must be erased before any new writes are issued to the same or a referenced page.

SCC Garbage Collection. With SCC Invariant, any committed intention record can be garbage collected as long as a newer version of the same logical page is committed. Any uncommitted intention record can be garbage collected at any time. Garbage collection involves selecting a candidate block, copying the valid pages out of it, and erasing the block to add to the free-blocks list. TxFlash copies each valid version to another location, preserving the same metadata. If the system crashes after copying a version and before erasing the block, multiple identical versions may be present for the same page. This is a minor complication. TxFlash can pick one copy as the principal copy and treat the others as redundancies to be erased when convenient.

SCC Recovery. During recovery, SCC classifies the intention records and identifies the highest committed version for each logical page, as follows:

Since isolation is guaranteed, *i.e.*, there are no overlapping write operations for the same page, for each logical page, the recovery algorithm only has to choose between the intentions having the *highest* and the *second highest* version numbers. This is true for the following reason. The intentions having the second highest version numbers must have been committed, since the application must have completed their transactions before going on to start a subsequent transaction on the same page. The only question to answer is whether the highest version numbered intention is also committed for a page.

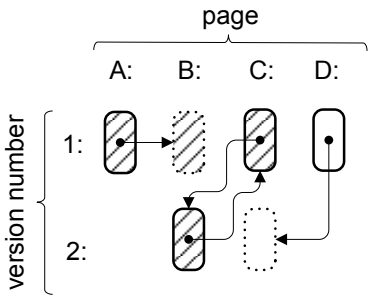


Figure 4: **An Example TxFlash System State with SCC.**

Let h_P represent the highest version number for a page P and P_h be the intention record with the highest version number. The goal of the recovery algorithm is to determine, for every page P, whether P_h is committed or uncommitted using the following analysis. Let $Q_l = P_h.next$. These values are available from the metadata portion of the P_h intention record. Let h_Q be the highest version number of any intention that exists on the storage device for page Q. There are three cases.

I. $h_Q > l$: P_h is a committed intention because Q_l is committed because of the presence of h_Q (SCC Invariant), and so is P_h (Cycle Property). For example, consider A_1 in Figure 4, whose next-link is B_1. Since the highest version for B is B_2, B_1 is committed and therefore A_1 is committed too.

II. $h_Q < l$: P_h is an uncommitted intention. The reasoning is as follows: the transaction involving P_h could not have completed, because if it had, there would be an intention of page Q with a version number at least as high as l. Consider D_1 in Figure 4, whose next-link C_2 is greater than the highest version numbered intention C_1 for page C. Therefore, D_1 is uncommitted.

III. $h_Q = l$: P_h links to another highest version numbered intention Q_h, and the answer is the same as for the intention Q_h, which may be determined recursively. If this results in a cycle, then all of the involved intentions are committed intentions (Cycle Property). For example, in Figure 4, following the next-link from B_2, a cycle is detected and B_2 is classified as committed.

For each page, the last committed intention is identified using the above analysis and the remap table is updated accordingly. Since each logical page is visited only once and only the top two versions are considered for each logical page, the running time of the SCC recovery takes $O(n)$, where n is the number of logical pages.

3.3 Back Pointer Cyclic Commit (BPCC)

The SCC has the advantage that it can be implemented relatively easily with minimal changes to the recovery and garbage collection activities normally performed by an SSD. The simplicity of SCC hinges on the SCC Invariant, which necessitates the erasure of uncommitted intentions before newer versions can be created. The needed erasure could add to the latency of the writes. We introduce Back Pointer Cyclic Commit (BPCC), a variation of the SCC, that does not require such erasures.

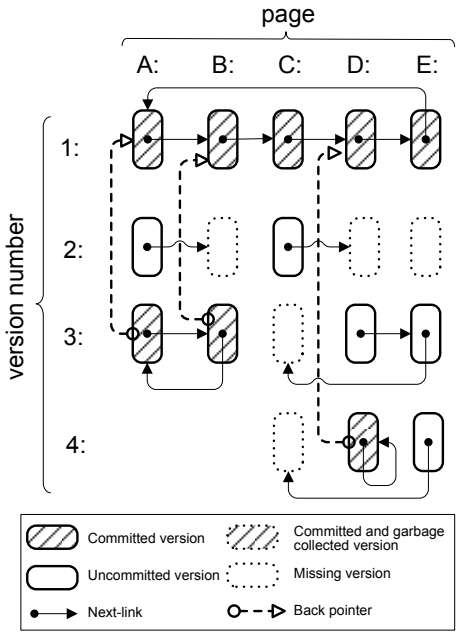


Figure 5: **An Example TxFlash System State with BPCC.**

BPCC indicates the presence of uncommitted intention records by adding more information to the page metadata. Specifically, the intention record r for a page also stores the last committed version number for that page through a *back pointer*, $r.back$. That is, before writing an intention P_k of a page P, in addition to the identity of the page and the next-link, a pointer to the last committed version of P, say P_i (where $i < k$), is also stored. Typically, the last committed version number will be the version number immediately previous to the version number of the intention (*i.e.*, $i = k-1$). This last committed version number provides enough information to determine whether uncommitted intentions are left behind by aborted transactions. Specifically, if there exists a $P_j \in \mathcal{S} \cup \mathcal{R}$, where $i < j < k$, then P_j must be uncommitted. Figure 2(c) shows the necessary additions to the metadata to include the back pointer.

For any intention record P_j, an intention record P_k with $P_k.back = P_i$ satisfying $i < j < k$ is a *straddler* of P_j as it straddles the uncommitted intention P_j. It is important to notice that a committed intention can never be straddled. For correct operation, BPCC upholds the following BPCC Invariant:

BPCC Invariant: For a highest version numbered intention record $P_h \in \mathcal{S}$, let $Q_l = P_h.next$. If there

exists a $Q_k \in \mathcal{S}$ with $k > l$ and there exists no straddler for Q_l, then P_h is committed.

BPCC Initialization. When TxFlash starts for the first time, it initializes the metadata for each page by setting the version number to 0, the next-link to itself, and the back pointer to itself.

Handling Uncommitted Intention Records. If an intention record P_i belongs to an aborted transaction or is classified as uncommitted during recovery, a newer version of P can be written with a back pointer to the committed version lower than P_i, effectively straddling P_i.

Figure 5 shows an example system state as updated by BPCC, where the back pointers are shown as dashed arrows. Consider the transaction updating A_3 and B_3. When the transaction is in progress, the last committed version of A is A_1 and therefore A_3 stores a pointer to A_1. This back pointer provides the proof that the intermediate version A_2 is uncommitted and, moreover, that any highest version numbered intention with a next-link pointer that refers to A_2 must also be uncommitted. Notice that, unlike SCC, BPCC can proceed with writing A_3 and B_3 without erasing the uncommitted intentions.

BPCC Garbage Collection. To preserve BPCC Invariant, before an intention record r can be garbage collected, the protocol must make sure that the status of the other intention records straddled by r is no longer useful during recovery. We capture such information by introducing a *straddle responsibility set* (SRS) for each intention record, as follows:

Given an intention record $P_k \in \mathcal{S}$ with $P_k.back = P_i$, the straddle responsibility set of P_k is

$$\{Q_l \in \mathcal{S} \mid Q_l.next = P_j \; and \; i < j < k\}.$$

For each Q_l in P_k's straddle responsibility set, the fact that it is uncommitted hinges on the availability of P_k's back pointer. Therefore, P_k can be garbage collected *only* after all such Q_l are erased. More precisely, for BPCC, an intention record P_i can be garbage collected if and only if $SRS(P_i)$ is empty and P_j for some $j > i$ is committed.

Various optimizations exist to remove entries from the straddle responsibility sets. This makes it possible to have certain intention records garbage collected earlier. We list some of them here.

- For any $Q_l \in SRS(P_k)$, if a higher version of Q is committed, then Q_l can be removed from $SRS(P_k)$. This is because Q_l can never in the future be the highest numbered version, once a later version is committed. Observe in Figure 5 that version B_3 straddles the next-link of A_2 but A_3 is a more recent committed version. So, A_2 can be removed from $SRS(B_3)$.
- If $Q_l \in SRS(P_j) \cap SRS(P_k)$, and $j < k$, Q_l can be removed from $SRS(P_j)$. This is because a higher version P_k straddles Q_l as well. In Figure 5, D_3 straddles the next-link of C_2, but D_4 is a later version of D and also straddles the next-link of C_2, so C_2 can be removed from $SRS(D_3)$.
- If P_j belongs to an aborted transaction and no higher version of P is committed, then $SRS(P_j)$ can be set to empty. This is because even if P_j is garbage collected, any uncommitted version of P above the highest committed version can be classified correctly, either by following its next-link or by another straddler of P with a higher version. In Figure 5, version E_4 straddles the next-link of D_3 but E_4 is later than the last committed version on page E so $SRS(E_4)$ can be set to empty.

BPCC Recovery. For BPCC, the recovery algorithm works by determining whether the highest-numbered version for a given page is committed or not. If it is committed, then it is the last committed version; otherwise the version itself indicates the last committed version through the back pointer, and all the intermediate versions between the highest-numbered version and its back-pointer version must be uncommitted.

For every page P, let P_h be the intention record with the highest version number, $Q_l = P_h.next$, and Q_h be the highest version numbered intention present for Q. The commit status of P_h is determined by the following analysis. There are three cases.

I. $h_Q > l$: In this case, check if Q_l is an uncommitted version by looking for a straddler. That is, look for some Q_i such that $i > l$ and $Q_i.back < l$. If such a straddler is present, then P_h is uncommitted, else P_h is committed (BPCC Invariant). In Figure 5, consider the next-link of C_2, which is D_2. Since the highest version numbered intention D_4 is greater than D_2 and since D_4 straddles D_2, C_2 can be decided as uncommitted.

II. $h_Q < l$: P_h is uncommitted and the reasoning is similar to the case (**II**) of SCC recovery. Consider the next-link of E_4, which is C_4 in Figure 5. Since C_4 is greater than the highest version for C (*i.e.*, C_2), E_4 is uncommitted.

III. $h_Q = l$: The next-link must be a highest-numbered version and its commit status can be determined by recursive analysis. The recursive application must check for a complete cycle, which indicates that all of the involved versions are committed (Cycle Condition). The cycle between A_3 and B_3 in Figure 5 is an example of this case.

3.4 Discussion

Dynamic Transactions. Transactions as implemented in most journaling file systems are static, *i.e.*, the pages to write are known before the transaction begins. But in general, transactions are dynamic, *i.e.*, all the writes is-

	TC	SCC	BPCC
Metadata/page	16 bytes	24 bytes	28 bytes
Space overhead	1 commit/tx	None	None
Perf. overhead	1 write/tx	None	None
Garbage collection	Simple	Simple	Complex
Recovery	Simple	Simple	Complex
Aborted transactions	Leave	Erase	Leave

Table 1: **Protocol Comparison.** *Traditional commit compared with the cyclic commit protocol variants.*

sued inside a transaction are not known before the transaction begins and they are determined only during the course of the execution. It is important to note that cyclic commit can support this more general semantics. For cyclic commit to work correctly, it is *not* necessary to know all the pages that are written within a transaction. For the next-link pointers to be filled correctly in the cycle, it is sufficient to know the next page write in the transaction. This is achieved simply by holding the current write in a buffer until the next write is issued by the transaction. When the last page is written, it is linked back to the first write of the transaction.

TxFlash for Databases. In addition to file systems, databases can benefit from the transactional support in a storage system. However, there are certain challenges that must be overcome. First, databases require the generic transactional interface with reads and writes instead of a simple `WriteAtomic` model. Second, in our current implementation we use a simple lock-based isolation technique, which may not be sufficient for a database system. We are exploring the use of a more fine-grained concurrency control mechanism that can take advantage of the multiple page versions in an SSD [18]. In addition, we may need a deadlock detection and abort mechanism in TxFlash for databases.

Metadata Overhead. In Table 1, we present a qualitative comparison of traditional commit with the two cyclic commit variants. We assume 4 bytes for transaction ID, 4 bytes for version number, and 8 bytes for logical page number.

One possible concern in cyclic commit is the space and write overheads imposed by additional pointers stored in the metadata portion of an intention record. As shown in Table 1, we need a maximum of 28 bytes to store the cyclic commit specific metadata. This still leaves enough space for larger ECCs to be stored in the future. Regarding the write overhead, the hardware specification of the Samsung flash package [21] states that when a flash memory page is written, it is recommended to write the entire 4 KB of data *and* 128 bytes of metadata to maintain correctly the on-chip error detection and correction codes. Therefore, the overhead of writing additional metadata is common for all the protocols even though the traditional commit uses less space than cyclic commit.

3.5 Summary

SCC requires that the uncommitted intentions are erased before the system updates the corresponding pages, whereas BPCC demands more metadata to be kept in each intention record and a more complicated analysis in the recovery algorithm, but it does not require the erasure of any uncommitted intentions. BPCC also requires that the intention records containing obsolete versions be reclaimed according to a certain precedence order, whereas SCC has no such requirement. Neither protocol requires any checkpointing (or reorganization) overhead for garbage collection. Depending on the overhead of erasing a storage page, the expected rate of failures and updates, and the space available to store metadata, either SCC or BPCC may be preferred.

4 Implementation

In this section, we present the implementation details of the TxFlash simulator, a pseudo-device driver, and the modified TxExt3 file system.

4.1 TxFlash Simulator

We modify the trace-driven SSD simulator from previous work [1] to develop the TxFlash simulator. The SSD simulator itself is extended from the DiskSim simulator [4] by adding an SSD model into the framework. The SSD simulator can evaluate various flash package configurations, interleaving, wear-leveling, and garbage collection techniques. The simulator maintains an in-memory remap table and a free-blocks list to process requests; the requests are stored in a per-flash-package queue. In order to reduce the recovery time, the simulator also writes a per-block summary page. During recovery, instead of reading every page, only the summary pages are read.

WriteAtomic Interface. We built our TxFlash simulator to support the `WriteAtomic` interface, a restricted form of transaction where pages written are known before the transaction commit begins. File systems issue a `WriteAtomic` command in which they pass all the page numbers of the transaction writes and receive a transaction ID; then they issue the data for each transactional page write. The simulator also supports an `Abort` command in addition to the regular `read` and `write` commands.

Other Modifications. The TxFlash simulator keeps track of the list of pages modified by an in-progress transaction to ensure isolation. This list is cleared when the transaction completes or aborts. Even though each individual page write is atomic, a block erase operation is not. That is, TxFlash can crash during a block erase, which may leave the block contents in an unpredictable

state. The page integrity can be verified if checksums are stored in its metadata, and the simulator models a 16 byte checksum per physical page. Because of its hardware limitations, TxFlash supports only bounded transactions (this is not a limitation of the cyclic commit), and our simulator is configured to support transactions of maximum size 4 MB (*i.e.*, containing up to one thousand 4 KB pages).

TxFlash supports a regular write operation because journaling file systems such as Ext3 issue single page writes to file system super blocks outside a transaction even in data journaling mode. TxFlash treats the regular `write` operation to a single page as a trivial transaction, *i.e.*, a transaction containing that single write, and this can be implemented with no additional costs.

Traditional Commit in TxFlash Simulator. To evaluate the trade-offs, we also implemented the traditional commit protocol in the TxFlash simulator. It works by writing one commit record for every transaction that modifies more than one page. Since there is no log reorganization, the commit record is preserved until all the page versions become obsolete, and the commit record serves as a proof as to whether a page version is committed or not. In order to do a fair evaluation, we tuned our traditional commit implementation to be efficient for single-page transactional write, which is handled by just writing the page version with a bit set in its metadata to indicate that it is a single-page transaction without any additional commit record. That is, for a single page write, traditional commit is as efficient as cyclic commit.

We simulate a 4 KB commit record write because the smallest write-granularity on modern NAND-based flash packages is 4 KB (similar to the 512 bytes granularity on disks). However, commit records are typically small, storing the transaction ID and a commit identifier. Therefore, simulating a 4 KB write adds unnecessary overhead. If SSD controllers support other types of byte-addressable memory such as battery-backed RAM or NOR-based flash, then commit records can be stored more efficiently on them. In such cases, the overhead in traditional commit is likely to be very small. Since there is no log reorganization, recovery in traditional commit scans through the stable storage and finds the most recent version with a commit record for each page.

4.2 TxFlash Pseudo-Device Driver

We implement a pseudo-device driver supporting the new transactional calls, `WriteAtomic` and `Abort`, through the `ioctl` interface. The driver associates a unique transaction ID with each transaction and forwards the read and write requests to the underlying storage device. A tracing framework is also built within the driver to generate traces for the TxFlash simulator. We record attributes such as page number, I/O type (read/write), time stamp, I/O size, and transaction ID in the trace.

New transactional commands, when issued, may cause a small overhead. For example, when a file system issues a `WriteAtomic` command, the page numbers from the command are copied to the TxFlash buffer and the new transaction ID generated by the TxFlash is copied back to the file system. The driver emulates such overheads to a certain extent by copying data in the system memory.

4.3 TxExt3

Our initial goal was to build the transaction support inside Ext2 with the help of TxFlash. However, this requires the abstraction of an in-core transaction that can buffer writes for write-back caching. The journaling module (jbd) of Ext3 already provides this abstraction (in addition to the redo logging facility) and therefore, instead of re-implementing the transaction abstraction in Ext2, we reuse the jbd module and the Ext3 file system.

We modify the jbd module to use the `WriteAtomic` interface to create the TxExt3 file system. Instead of writing to the journal, the commit function uses the transactional commands from the pseudo-device driver. In TxExt3, there is no separate checkpointing process as in Ext3. Therefore, once all the transactional writes are over, TxExt3 releases the in-core buffers and proceeds with the normal operations.

5 Evaluation

We evaluate our design and system through several stages: first, we show that cyclic commit outperforms traditional commit, both in terms of performance and space overheads (§5.1); second, we compare TxFlash against an SSD to estimate the overheads of transactional support (§5.2); and finally, we run benchmarks on top a real SSD to study the end-to-end file system performance improvement (§5.3). Throughout this section, by transaction we refer to the transactional writes as issued by the TxExt3 file system during its journal commit.

Workloads and Simulator Settings. We collect traces from TxExt3 under both data and metadata journaling modes by mounting it on top of the pseudo-device driver and using the driver's tracing framework. We run a variety of benchmarks: IOzone [14] is a complex microbenchmark suite and it is run in auto mode with a maximum file size of 512 MB; Linux-Build is a CPU intensive workload and it copies, unpacks, and builds the entire Linux 2.6.18 source tree; Maildir simulates the widely used maildir format for storing e-mails [2] and we run it with a distribution of 10,000 emails, whose sizes vary from 4 KB to 1 MB; TPC-B [27] simulates

a database stress test by issuing 10,000 credit-debit-like operations on the TxExt3 file system. The TxExt3 file system issues one transaction at a time, so there is little concurrency in the workload. In our experiments, the transaction size varies among the benchmarks and is determined by the sync intervals. Since IOzone and Linux-build do not issue any sync calls, their transaction sizes are quite large (over 2 MB); Maildir and TPC-B issue synchronous writes and therefore result in smaller transactions (less than 100 KB).

Ext3 provides only a limited transaction abort functionality. After a transaction abort, Ext3 switches to a fail-stop mode where it allows only read operations. To evaluate the performance of the cyclic commit under transaction aborts, we use a synthetic workload generator, which can be configured to generate a wide variety of transactions. The configuration parameters include the total number of transactions, a distribution of transaction sizes and inter-arrival times, maximum transaction concurrency, percentage of transaction aborts, and a seed for the random generator.

We configure our simulator to represent a 32 GB TxFlash device with 8 fully-connected 4 GB flash packages and use the flash package parameters from the Samsung data sheet [21]. The simulator reserves 15% of its pages for handling garbage collection. I/O interleaving is enabled, which lets the simulator schedule up to 4 I/O operations within a single flash package concurrently.

Model Checking. We verify the cyclic commit algorithm by specifying the SCC and BPCC protocols in TLA+ and checking them with the TLC model checker [10]. Our specifications model in-progress updates, metadata records, page versions, aborts, garbage collection, and the recovery process, but not issues such as page allocation or I/O timing. Our specifications check correctly up to 3 pages and 3 transactions per page; state explosion prevents us from checking larger configurations. This work is published elsewhere [19].

5.1 Cyclic Commit vs. Traditional Commit

In order to find the real performance benefits of the commit protocols, we must compare them only under transactional writes, as all of them work similarly under non-transactional writes. Therefore, in the following section, we only use traces from the data journaling mode or from the synthetic transaction generator.

Impact of Transaction Size. First, we compare cyclic commit with traditional commit to see whether avoiding one write per transaction improves the performance. The relative benefits of saving one commit write must be higher for smaller transactions. Figure 6 compares the transaction throughput of BPCC and TC under different transaction sizes. The TxFlash simulator is driven with the trace collected from a sequential writer, which varies its `sync` interval to generate different transaction sizes. Note that the performance numbers are the same for both SCC and BPCC, as they differ only when there are aborted transactions. For small transactions, all the page writes can be simultaneously issued among multiple flash packages and therefore, while BPCC takes time t to complete, TC takes $2t$ because of the additional commit write and write ordering, and this leads to a 100% improvement in transaction throughput. From the Figure, we observe that the performance improvement is about 95% when the transaction is of the order of 100 KB, and drops with larger transactions. Even larger transactions benefit from BPCC, for example, throughput improves by about 15% for transactions of size 1000 KB. For single page transactions, both the protocols perform similarly (not shown).

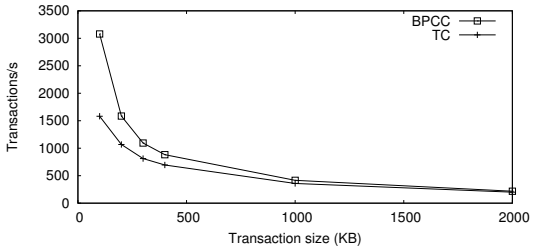


Figure 6: **Impact of Transaction Size.** *Transaction throughput vs. transaction size. TC uses a 4 KB commit record.*

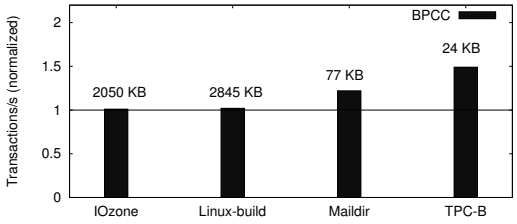


Figure 7: **Performance Improvement in Cyclic Commit.** *Transaction throughput in BPCC, normalized with respect to the throughput in TC. The throughput of IOzone, Linux-build, Maildir, and TPC-B in TC are 31.56, 37.96, 584.89, and 1075.27 transactions/s. The average transaction size is reported on top of each bar.*

Performance Improvement and Space Overhead. Next, we compare the commit protocols under macro benchmarks. Figure 7 plots the transaction throughput in TxFlash with BPCC and it is normalized with respect to the throughput in TC under various macro benchmarks. Since IOzone and Linux-build run asynchronously with large transactions, BPCC does not offer any benefit (less than 2% improvement in transaction throughput). On the other hand, Maildir and TPC-B stress the storage system with a large number of small transactions that cause high commit overhead in TC; under these cases, BPCC offers about 22% and 49% performance improvement for Maildir and TPC-B respectively. SCC performs similarly

	IOzone	Linux-build	Maildir	TPC-B
Space overhead	0.23%	0.15%	7.29%	57.8%

Table 2: **Space Overhead in Traditional Commit.** *Space overhead (ratio of the number of commit to the number of valid pages) for different macro benchmarks under TC.*

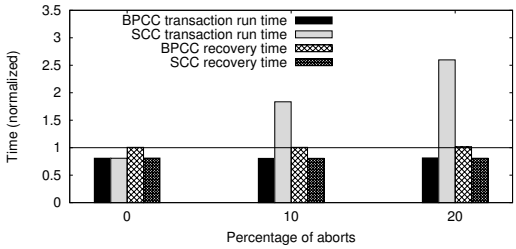


Figure 8: **Performance Under Aborts.** *Transaction time and recovery time of cyclic commit protocols under different percentages of transaction aborts. Y-axis is normalized with respect to the corresponding time in TC, which is around 1.43 ms and 2.4 s for the transaction time and recovery time, respectively.*

		TxFlash		
	SSD	+TC	+SCC	+BPCC
LOC	7621	9094	9219	9495

Table 3: **Implementation Complexity.** *Lines of code in SSD and TxFlash variants.*

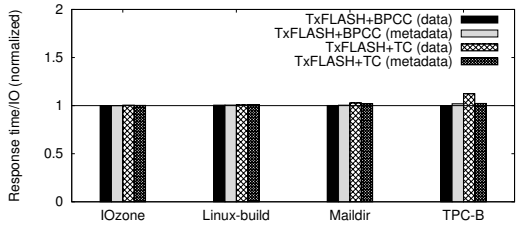


Figure 9: **TxFlash Overhead.** *I/O response time of BPCC and TC, normalized with respect to that of an SSD. The I/O response times of IOzone, Linux-build, Maildir, and TPC-B in an SSD are (0.72, 0.71), (0.70, 0.68), (0.59, 0.62), and (0.42, 0.39) ms in data and metadata journaling modes.*

under all workloads.

Table 2 presents the space overhead due to the additional commit record in TC. The space overhead is measured as the ratio of the number of commit pages to the valid pages in the system. The space overhead can increase significantly if there are small transactions updating different pages in the system (*e.g.*, Maildir and TPC-B). For large transactions, this overhead is quite small as evident from the IOzone and Linux-build entries.

Performance Under Aborts. Our next experiment compares the commit protocols under transaction aborts. We use the synthetic workload generator to create 20,000 transactions, each with an average size of 250 KB and measure the normal and recovery performance under different degrees of transactions aborts. Figure 8 presents the results, where for the normal performance we plot the average transaction time, and for the recovery we plot the time to read the stable storage and find the consistent versions. During recovery, only the per-block summary pages are scanned from the stable storage. The results are normalized with respect to TC.

From Figure 8, we draw the following conclusions. First, because SCC must erase an aborted page before letting another transaction write to the same logical page, its performance suffers as transaction aborts increase. BPCC does not incur any such overhead. Second, SCC has better recovery time than BPCC and TC because during recovery it considers only the top 2 versions for every page rather than paying the cost of analyzing all the page versions. In the presence of aborts and failures, the recovery time of BPCC also includes the time to find the appropriate straddle responsibility sets. This results in a small overhead when compared to TC (less than 1%). The recovery time can be improved through several techniques. For example, TxFlash can periodically checkpoint the remap table in the flash memory.

Protocol Complexity. Although beneficial, cyclic commit, specifically BPCC, is more complex than TC. In Table 3, we list the lines of code (LOC) for the regular SSD and TxFlash with different commit protocols. Treating the LOC as an estimator of complexity, TxFlash adds about 25% additional code complexity to the SSD firmware. Among the three commit protocols, TC is relatively easier to implement than the other two; BPCC is the most complex and most of its complexity is in the recovery and garbage collection modules.

5.2 TxFlash vs. SSD

Our next step is to measure the overhead of TxFlash when compared to a regular SSD under the same workloads. We use the traces collected from TxExt3 and run them on the TxFlash and SSD simulators. When running on the SSD simulator, we remove the `WriteAtomic` calls from the trace. Note that an SSD does not provide any transactional guarantees to the TxExt3 traces and we just measure the read-write performances.

Performance Overhead. Figure 9 presents the average I/O response time of TxFlash and SSD under various workloads in both data and metadata journaling modes. We configure TxFlash to run BPCC, but it performs similarly under SCC. From the Figure, we can notice that TxFlash with BPCC imposes a very small overhead (less than 1%) when compared to a regular SSD, essentially offering the transactional capability for free. This small overhead is due to the additional `WriteAtomic` commands for TxFlash. However, in TxFlash with TC, the additional commit writes can cause a noticeable performance overhead, especially if there are a large number of small transactions; for example, in Maildir and TPC-B under data journaling, TxFlash with TC incurs an additional overhead of 3% and 12%, respectively.

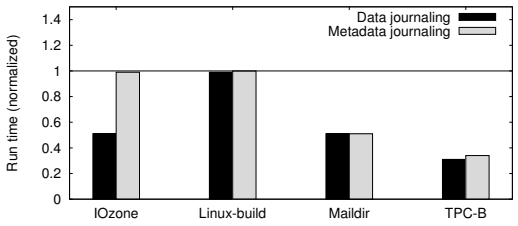


Figure 10: **End-to-End Performance.** *Benchmark run times on TxExt3-with-TxFlash under the data and metadata journaling modes, normalized with respect to their corresponding run times in Ext3-with-SSD. IOzone, Linux-build, Maildir, and TPC-B take (94.16, 43.38), (267.51, 264.40), (115.77, 146.79), and (82.73, 121.73) seconds in data and metadata journaling modes on Ext3.*

Memory Requirements. An SSD stores its remap table, free-blocks list, block-specific, and package-specific metadata in volatile memory (this is in addition to the memory that may be used for read or write buffering). For our configuration, an SSD requires about 4 MB per 4 GB flash package for its volatile structures. In addition, TxFlash needs memory to keep track of the in-progress and aborted transactions, to store the extra metadata per page, and to maintain the straddlers (only in BPCC). This requirement can vary depending on the maximum size of a transaction, the number of concurrent transactions, and the number of aborts. For a 4 GB flash package, to support a maximum of 100 concurrent transactions, with each having a maximum size of 4 MB, and an average of 1 abort per 100 transactions, we need an additional 1 MB of memory per 4 GB flash package. That is, for this configuration, TxFlash need 25% more memory than a regular SSD to support transactions.

5.3 End-to-End Improvement

While we use the simulator to understand the device-specific characteristics, we want to find out the end-to-end file system performance when running on a TxFlash. We run the pseudo-device driver on top of a 32 GB real SSD and export the pseudo-device to TxExt3 and Ext3. All our results are collected with the SSD cache disabled. Our previous evaluation from §5.2 shows that TxFlash (running either BPCC or SCC) adds little overhead when compared to SSD, and even this small overhead is emulated by the pseudo-device driver. For Ext3, the pseudo-device driver just forwards the I/O requests to the SSD.

We run the benchmarks on TxExt3 and Ext3 mounted on the pseudo-device under both data and metadata journaling, and the results are presented in Figure 10, which plots the run time of each benchmark normalized with respect to the corresponding run time on Ext3. TxExt3 with TxFlash outperforms Ext3 for two reasons: first, on each transaction, a commit write is saved and this can result in large savings for small transactions, even in metadata journaling (for example, in Maildir and TPC-B); second, Ext3 performs checkpointing, where it rewrites the information from the log into its fixed-location, and for data journaling this overhead can be significant (for example, in IOzone). Both the absence of commit write and checkpointing combine to reduce the run time by as much as around 65% (for TPC-B). However, Linux-build is compute intensive and the improvements are less than 1% because the transactions are large and most of the checkpointing happens in the background.

File system complexity can be reduced by using the transactional primitives from the storage system. For example, the journaling module of TxExt3 contains about 3300 LOC when compared to 7900 LOC in Ext3. Most of the reduction were due to the absence of recovery and revoke features and journal-specific abstraction.

5.4 Discussion and Summary

Another possible evaluation would be to compare a file system implementing cyclic commit and running on a regular SSD with TxExt3 running on TxFlash. This would let us find out if there are performance benefits in keeping the transactional features in the file system.

However, we face several limitations in building the cyclic commit inside a file system. First, current SSDs do not export the metadata portion of physical pages. As a result, cyclic commit may not be implemented as efficiently as described in this paper and therefore, the comparison would not be meaningful. Second, SSDs do not expose their garbage collection policies and actions. But, in BPCC, it is important to collect the obsolete pages in certain order and unfortunately, this control is not available to the file systems. Finally, if cyclic commit is implemented in a file system, it must use a variant of CoW and as a result, multiple indirection maps will be present in the system (one in the file system and the other in the SSD) that may lead to performance and space overheads.

In summary, we derive the following conclusions. First, in comparison with traditional commit, cyclic commit has the potential to improve the transaction throughput (by as much as 100%) and reduce the space overhead for small transactions, while matching the traditional performance for large transactions. Second, TxFlash with cyclic commit can provide transactional features with negligible overhead. Finally, a file system running on TxFlash can eliminate the write ordering problem and cut down the number of writes to half, resulting in large improvements for I/O intensive workloads.

6 Related Work

Mime [5] provides transaction support on disk drives using shadow copies. Mime offers the new function-

alities through visibility groups, which are used to ensure isolation between concurrent transactions. In addition to the standard `read` and `write` calls, Mime provides a richer set of APIs such as barriers. In contrast, TxFlash provides a simple `WriteAtomic` call, motivated by minimal complexity and file system support. Mime and TxFlash run on the storage controllers and share some of the implementation techniques, *e.g.*, CoW, block remapping, and recording data version in metadata. However, the underlying protocols are quite different. Mime uses the standard checkpointing and operation logging, whereas TxFlash uses cyclic commit.

Atomic recovery unit (ARU) [9] is an abstraction provided by the Logical Disk [6], which exports a general abstraction of a logical block interface to separate disk management from file system management. ARU operates at a higher level than TxFlash, which runs on the SSD controller. ARU allows both `read` and `write` operations in transactions (TxFlash, in contrast, supports only writes) and offers a more general isolation semantics for reads; however, the concurrency control for write operations must be implemented by the clients, whereas TxFlash provides isolation among multiple `WriteAtomic` operations. Blocks allocated under an uncommitted ARU must be identified and released during recovery, which is similar to the garbage collection requirements in TxFlash.

Stasis [22] is a library that provides a wide-range of highly flexible transactional storage primitives using WAL for applications that require transactional support but cannot use a database. Stasis is more flexible than TxFlash: it supports user-level operations, enables redo or undo logging, provides different concurrency control mechanisms, and supports atomic updates to large objects. We consider TxFlash and cyclic commit as complementary to Stasis. For example, Stasis can implement cyclic commit as one of the commit protocols.

One of the main differences between TxFlash and other disk-based transactional systems like Mime and ARU is that disk-based systems must reorganize the data for improved read performance, whereas TxFlash does not. In fact, it is harder to reorganize data in certain systems because the logical relationship between two disk blocks is not known at the disk controller.

Rio file cache is a recoverable area of main memory and Vista is a user-level library, and together, they provide light-weight transactional features that can be used to eliminate costlier synchronous disk I/O operations [11]. Rio Vista delivers excellent performance by avoiding the redo log and system calls and by using only one memory copy. TxFlash and Rio Vista operate at different layers of storage hierarchy; since Rio operates at a higher level (main memory), it works only for working sets that fit in main memory. Moreover, Rio does not provide isolation, while TxFlash offers this guarantee.

Park *et al.* [15] propose an atomic write interface for flash devices, taking advantage of the non-overwrite nature of flash pages. They store the transaction IDs on all the pages and write a commit record with the transaction ID to ensure atomicity. They modify the FAT file system to use the new interface and run it on top of an emulator. Since file system buffering can complicate the construction of transactions, their modified FAT file system runs synchronously. In contrast, we use the cyclic commit and modify Ext3, which already buffers transactions.

Transactional Flash File System (TFFS) [8] is built for NOR-based flash devices on embedded microcontrollers and provides transactional guarantees with a richer set of APIs. Unlike TxFlash, TFFS supports both reads and writes within a transaction, but write operations are all synchronous. Similar to other systems, TFFS uses an indirection called logical pointers and a variant of CoW called versioned tree structures to implement transactions. TFFS also allows non-transactional operations but does not guarantee any serializability.

Other flash-based file systems, such as JFFS2 [17] and YAFFS2 [12], have been designed for embedded devices. They use variations of log-structured design [20] and run directly on top of flash, sidestepping the FTL to avoid the cost of double-layering. Unlike TxFlash, which reads only the summary pages, JFFS2 scans the entire flash memory to rebuild its data structures; YAFFS2 carefully avoids this using checkpoints. None of these file systems provide transactional support.

Since file systems have higher-level semantic knowledge, *e.g.*, whether a page is free or not, they can do better garbage collection than a storage controller. Such information can be quite useful in TxFlash, not only in garbage collection, but also to quickly recover by not examining free pages.

7 Conclusion

In this paper, we revisit the concept of transactional support in the storage device in light of a new storage medium, the flash memory. The unique properties of the medium demand new designs in order to provide better performance and space benefits. The main contribution of this work is the novel cyclic commit protocol, which ensures atomicity by using the additional metadata on physical pages, thereby removing the overheads associated with a separate commit record. We design and implement two variants of the cyclic commit, SCC and BPCC, and both perform better than the traditional commit, especially for small transactions.

We learned a few things along the way. First, model checking our protocols helped us not only verify their correctness, but also understand *why* the protocols are

correct. Moreover, the model checker pointed out flaws in the alternative designs we investigated, which we would have missed otherwise. Second, actual implementation can bring out issues that are otherwise missed. For example, we came across complex interactions between garbage collection, block allocation, and in-progress transactions in our simulation (we did not model check some of them for simplicity reasons) and fixed those corner cases. Finally, we believe that hardware innovations can often bring new software designs; this is true in the case of cyclic commit, which was motivated by developing a commit protocol for flash memory.

8 Acknowledgments

We are grateful to our shepherd, Lorenzo Alvisi, for his valuable and detailed feedback on our paper. We also thank James Hamilton, the members of Microsoft Research Silicon Valley, and the anonymous reviewers for their excellent suggestions and comments.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, pages 57–70, June 2008.

[2] D. J. Bernstein. Using maildir format. http://cr.yp.to/proto/maildir.html.

[3] S. Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2000.

[4] J. S. Bucy and G. R. Ganger. The DiskSim Simulation Environment Version 3.0 Reference Manual. Technical Report CMU-CS-03-102, Carnegie Mellon University, January 2003.

[5] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: a high performance parallel storage device with strong recovery guarantees. Technical Report HPL-CSP-92-9rev1, HP Laboratories, November 1992.

[6] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The Logical Disk: A New Approach to Improving File Systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 15–28, December 1993.

[7] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pages 1–8, June 1984.

[8] E. Gal and S. Toledo. A Transactional Flash File System for Microcontrollers. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 89–104, April 2005.

[9] R. Grimm, W. C. Hsieh, W. de Jonge, and M. F. Kaashoek. Atomic Recovery Units: Failure Atomicity for Logical Disks. In *International Conference on Distributed Computing Systems (ICDCS '96)*, pages 26–37, May 1996.

[10] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[11] D. Lowell and P. Chen. Free transactions with Rio Vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 92–101, October 1997.

[12] C. Manning. YAFFS: Yet Another Flash File System. `http://www.aleph1.co.uk/yaffs`, 2004.

[13] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[14] W. Norcutt. The IOzone Filesystem Benchmark. http://www.iozone.org/.

[15] S. Park, J. H. Yu, and S. Y. Ohm. Atomic Write FTL for Robust Flash File System. In *Proceedings of the Ninth International Symposium on Consumer Electronics*, pages 155–160, June 2005.

[16] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 206–220, Oct 2005.

[17] Red Hat Corporation. JFFS2: The Journalling Flash File System. `http://sources.redhat.com/jffs2/jffs2.pdf`, 2001.

[18] D. P. Reed. *Naming and Synchronization in a Decentralized Computer System*. Ph.D. Dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[19] T. L. Rodeheffer. Cyclic Commit Protocol Specifications. Technical Report MSR-TR-2008-125, Microsoft Research, September 2008. ftp://ftp.research.microsoft.com/pub/tr/TR-2008-125.pdf.

[20] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[21] Samsung Corporation. K9XXG08XXM Flash Memory Specification. `http://www.samsung.com/global/system/business/semiconductor/product/2007/6/11/NANDFlash/SLC_LargeBlock/8Gbit/K9F8G08U0M/ds_k9f8g08x0m_rev10.pdf`, 2007.

[22] R. Sears and E. Brewer. Stasis: flexible transactional storage. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 29–44, 2006.

[23] M. Seltzer. *File System Performance and Transaction Support*. PhD thesis, EECS Department, University of California, Berkeley, Jun 1993.

[24] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '93)*, pages 307–326, January 1993.

[25] D. A. Solomon. *Inside Windows NT*. Microsoft Programming Series. Microsoft Press, 2nd edition, May 1998.

[26] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14, January 1996.

[27] Transaction Processing Council. TPC Benchmark B Standard Specification, Revision 3.2. Technical Report, 1990.

[28] S. C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[29] J. Yang, C. Sar, and D. Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 131–146, November 2006.

[30] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 273–288, December 2004.

Avoiding File System Micromanagement with Range Writes

Ashok Anand, Sayandeep Sen, Andrew Krioukov*, Florentina Popovici†,
Aditya Akella, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Suman Banerjee
University of Wisconsin, Madison

Abstract

We introduce *range writes*, a simple but powerful change to the disk interface that removes the need for file system micromanagement of block placement. By allowing a file system to specify a set of possible address targets, range writes enable the disk to choose the final on-disk location of the request; the disk improves performance by writing to the closest location and subsequently reporting its choice to the file system above. The result is a clean separation of responsibility; the file system (as high-level manager) provides coarse-grained control over placement, while the disk (as low-level worker) makes the final fine-grained placement decision to improve write performance. We show the benefits of range writes through numerous simulations and a prototype implementation, in some cases improving performance by a factor of three across both synthetic and real workloads.

1 Introduction

File systems micromanage storage. Consider placement decisions: although modern file systems have little understanding of disk geometry or head position, they decide the exact location of each block. The file system has coarse-grained intentions (e.g., that a data block be placed near its inode [22]) and yet it applies fine-grained control, specifying a single target address for the block.

Micromanagement of block placement arose due to the organic evolution of the disk interface. Early file systems such as FFS [22] understood details of disk geometry, including aspects such as cylinders, tracks, and sectors. With these physical characteristics exposed, file systems evolved to exert control over them.

The interface to storage has become more abstract over time. Currently, a disk presents itself as a linear array of blocks, each of which can be read or written [2, 21]. On the positive side, the interface is simple to use: file systems simply place blocks within the linear array, making it straightforward to utilize the same file system across a broad class of storage devices.

On the negative side, the disk hides critical information from the file system, including the exact logical-to-physical mapping of blocks as well as the current position of the disk head [32, 37]. As a result, the file system does not have accurate knowledge of disk internals. However, the current interface to storage demands such knowledge, particularly when writing a block to disk. For each write, the file system must specify a single target address; the disk must obey this directive, and thus may incur unnecessary seek and rotational overheads. The file system micromanages block placement but without enough information or control to make the decision properly, precisely the case where micromanagement fails [6].

Previous work has tried to remedy this dilemma in numerous ways. For example, some have advocated that disks remap blocks on-the-fly to increase write performance [7, 10, 34]. Others have suggested a wholesale move to a higher-level, object-based interface [1, 13]. The former approach is costly and complex, requiring the disk to track a large amount of persistent metadata; the latter approach requires substantial change to existing file systems and disks, and thus inhibits deployment. An ideal approach would instead require little modification to existing systems while enabling high performance.

In this paper, we introduce an evolutionary change to the disk interface to address the problem of micromanagement: *range writes*. The basic idea is simple: instead of specifying a single exact address per write, the file system presents a set of possible targets (i.e., a range) to the disk. The disk is then free to pick where to place the block among this set based on its internal positioning information, thus minimizing positioning costs. When the request completes, the disk informs the file system which address it chose, thereby allowing for proper bookkeeping. By design, range writes retain the benefits of the existing interface, and necessitate only small changes to both file system and disk to be effective.

Range writes make a more successful two-level managerial hierarchy possible. Specifically, the file system (i.e., the manager) can exert coarse-grained control over block placement; by specifying a set of possible targets, the file system gives freedom to the disk without relinquishing all control. In turn, the disk (i.e., the worker) is given the ability to make the best possible fine-grained decision, using all available internal information.

Implementing and utilizing range writes does not come without challenges, however. Specifically, drive scheduling algorithms must change to accommodate ranges efficiently. Thus, we develop two novel approaches to scheduling of range writes. The first, *expand-and-cancel scheduling*, integrates well with current schedulers but in-

*Now a Ph.D. student at U.C. Berkeley
†Now at Google

duces high computational overhead; the second, *hierarchical range scheduling*, requires a more extensive reworking of the disk scheduling infrastructure but minimizes computational costs as a result. Through simulation, we show that both of these schedulers achieve excellent performance, reducing write latency dramatically as the number of targets increases.

In addition, file systems must evolve to use range writes. We thus explore how range writes could be incorporated into the allocation policies of a typical file system. Specifically, we build a simulation of the Linux ext2 file system and explore how to modify its policies to incorporate range writes. We discuss the core issues and present results of running a range-aware ext2 in a number of workload scenarios. Overall, we find that range writes can be used to place some block types effectively (e.g., data blocks), whereas other less flexibly-placed data structures (e.g., inodes) will require a more radical redesign to obtain the benefits of using ranges.

Finally, we develop and implement a software-based prototype that demonstrates how range writes can be used to speed up writes to the log in a journaling file system. Our prototype transforms the Linux ext3 *write-ahead log* into a more flexible *write-ahead region*, and in doing so avoids rotational overheads during log writes. Under a transactional workload, we show how range writes can improve journaling performance by nearly a factor of two.

The rest of this paper is organized as follows. In Section 2, we discuss previous efforts and why they are not ideal. We then describe range writes in Section 3 and study disk scheduling in Section 4. In Section 5, we show how to modify file system allocation to take advantage of range writes, and then describe our prototype implementation of fast journal writing in Section 6. Finally, in Section 7, we conclude.

2 Background

We first give a brief tutorial on modern disks; more details are available elsewhere [2, 26]. We then review existing approaches for minimizing positioning overheads.

2.1 Disks

A disk drive contains one or more *platters*, where each platter *surface* has an associated disk head for reading and writing. Each surface has data stored in a series of concentric circles, or *tracks*. Within each track are the *sectors*, the smallest amount of data that can be read or written on the disk. A single stack of tracks at a common distance from the spindle is called a *cylinder*. Modern disks also contain RAM to perform caching.

The disk appears to its client, such as the file system, as a linear array of logical blocks; thus, each block has an associated logical block number, or LBN. These logical blocks are then mapped to physical sectors on the platters. This indirection has the advantage that the disk can lay out blocks to improve performance, but it has the disadvantage that the client does not know where a particular logical block is located. For example, optimizations such as zoning, skewing, and bad-block remapping all impact the mapping in complex ways.

The service time of reading or writing a request has two basic components: *positioning time*, or the time to move the disk head to the first sector of the current request, and *transfer time*, or the time to read/write all of the sectors in the current request. Positioning time has two dominant components. The first component is *seek time*, moving the disk head over the desired track. The second component is *rotational latency*, waiting for the desired block to rotate under the disk head. The time for the platter to rotate is roughly constant, but it may vary around 0.5% to 1% of the nominal rate. The other mechanical movements (e.g., head and track switch time) have a smaller but non-negligible impact on positioning time [27].

Most disks today also support *tagged-command queueing* [24], in which multiple outstanding requests can be serviced at the same time. The benefit is obvious: with multiple requests to choose from, the disk itself can schedule requests it sees and improve performance by using detailed knowledge of positioning and layout.

2.2 Reducing Write Costs

In this paper, our focus is on what we refer to as the *positioning-time problem* for writes; how do we reduce or eliminate positioning-time overheads for writes to disk? The idea of minimizing write time is by no means new [20]. However, there is as of yet no consensus on the best approach or the best division of labor between disk and file system for achieving this goal. We briefly describe previous approaches and why they are not ideal.

2.2.1 Disk Scheduling

Disk scheduling has long been used to reduce positioning overheads for writes. Early schedulers, built into the OS, tried to reduce seek costs with simple algorithms such as elevator and its many variants.

More recently, schedulers have gone beyond seek optimizations to include rotational delay. The basic idea is to reorganize requests to service the request with the *shortest positioning time first (SPTF)* instead of simply the request with the shortest seek time (SSTF). Performing rotationally-aware scheduling within the disk itself is relatively straightforward since the disk has complete and accurate information about the current location of the disk head and the location of each requested block. In contrast, performing rotationally-aware scheduling within the OS is much more challenging, since the OS must predict the current head position. As a result, much of the scheduling work has been performed through simulation [18, 28] or has been crafted with extreme care [17, 23, 36, 38]

More fundamentally, disk scheduling alone cannot completely eliminate rotational delays. For example, if too few requests exist in the scheduling queue, smart scheduling cannot avoid rotation.

2.2.2 File System Structures

Another approach to solving the positioning-time problem for writes is to develop a file system that transforms the traffic stream to avoid small costly writes by design. A prime example is the Log-structured File System (LFS) [25]; many commercial file systems (e.g., WAFL [14], ZFS [31]) have adopted similar approaches.

LFS buffers all writes (including data and metadata) into *segments*; when a segment is full, LFS writes the segment in its entirety to free space at the end of the log. By writing to disk in large chunks (a few megabytes at a time), LFS amortizes positioning costs.

By design, LFS avoids small writes and thus would seem to solve the positioning-time problem. However, LFS is not an ideal solution for two reasons. First, this approach requires the widespread adoption of a new file system; history has shown such adoption is not straightforward. Second, LFS and similar file systems do not perform well for all workloads; in particular, underneath transactional workloads that frequently force data to disk, LFS performance suffers [29].

2.2.3 Write Indirection

Many researchers have noted that another way to minimize write delay is to appropriately control the placement of blocks on disk. This work, which introduces a layer of indirection between the file system and disk, can be divided into two camps: that which assumes the traditional interface to disk (an array of blocks), and that which proposes a new, higher-level interface (usually based on objects or similar abstractions).

Traditional Disks: In the first approach, the disk itself controls the layout of logical blocks written by the file system onto the physical blocks in the disk. The basic approach has been to perform *eager writing*, in which the data is written to the free disk block currently closest to the disk head. There are three basic problems with these approaches. First, this approach assumes that an *indirection map* exists to map the logical block address used by the file system to its actual physical location on disk [7, 10, 34]. Unfortunately, updating the indirection map atomically and recovering after crashes can incur a significant performance overhead. Second, these systems need to know which blocks are free versus allocated. Unfortunately, although the file system readily knows the state of each logical block, it is quite challenging for disks to know whether a block is live or dead [30]. Third, this approach forces the file system to completely relinquish any control over placement; given that the file system knows which blocks are related to one another and thus are likely to exhibit temporal locality (e.g., the inode and all data blocks of the same file), the file system would like to ensure that those blocks are placed somewhat near one another to optimize future reads. Thus, pushing full responsibility for block placement into the disk is not the best division of labor.

New Interfaces: A related set of efforts allows the disk to control placement but requires a new interface; this idea has appeared in different forms in the literature as Logical Disks [8], Network-Attached Storage Devices [13], and Object-based Storage [1]. With this type of new interface, the disk controls exactly where each object is placed, and thus can make intelligent low-level decisions. However, such an approach also has its drawbacks. First, and most importantly, it requires more substantial change of both disks and the clients that use them, which is likely a major impediment to widespread acceptance. Second, allowing the disk to manage objects (or similar constructs) implies that the disk must now be concerned with consistent update. Consider object-based storage: when adding a new block to an object, both the new block and a pointer to it must be allocated inside the disk and committed in a consistent fashion. Thus, the disk must now also include some kind of logging machinery (perhaps to NVRAM), duplicating effort and increasing the complexity of the drive. Logical disks go a step further, adding a new "atomic recovery unit" interface to allow for arbitrary writes to be grouped and committed together [8]. In either approach, complexity within the disk is increased.

2.3 A Cooperative Approach

Previous Approach	Problems
Disk scheduling [17, 18, 23, 28, 36, 38]	Needs many requests, hard to implement in OS
Write-anywhere file systems [14, 25, 31]	Gaining acceptance, synchronous workloads
Eager writing [10, 34]	Drive complexity, lack of FS information
Higher-level interfaces [1, 8, 13]	Drive complexity, gaining acceptance

In contrast to previous approaches, range writes divide the responsibilities of performing fast writes across both file system and disk; this tandem approach is reminiscent of scheduler activations [3], in which a cooperative approach to thread scheduling was shown to be superior to either a pure user-level or kernel-level approach. The file system does what it does best: make coarse-grained layout decisions, manage free space, track block ownership, and so forth. The disk does what it does best: take advantage of low-level knowledge (e.g., head position, actual physical layout) to improve performance. The small change required does not greatly complicate either system or significantly change the interface between them, thus increasing the chances of deployment.

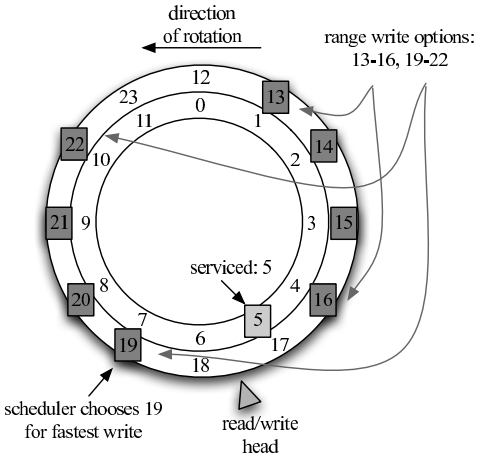


Figure 1: **Range Writes.** *The figure illustrates how to use range writes. The disk has just serviced a write to block 5, and is then given a write with two ranges: 13 through 16, and 19 through 22. The disk, given its current position, realizes that 19 will result in the fastest write, and thus chooses it as the target. The file system is subsequently informed.*

3 Range Writes

In this section, we describe range writes. We describe the basic interface as well as numerous details about the interface and its exact semantics. We conclude with a discussion of the counterpart of range writes: range reads.

3.1 The Basic Interface

Current disks support the ability to write data of a specified length to a given address on disk. With range writes, the disk supports the ability to write data to one address out of a set of specified options. The options are specified in a *range descriptor*. The simplest possible range descriptor is comprised of a pair of addresses, B_{begin} and B_{end}, designating that data of a given length can be written to any contiguous range of blocks fitting within B_{begin} and B_{end}. See Figure 1 for an example.

When the operation completes, the disk returns a *target address*, that is, the starting address of the region to which it wrote the data. This information allows the file system to record the address of the data block in whatever structure it is using (e.g., an inode).

One option the client must include is the *alignment* of writes, which restricts where in the range a write can be placed. For example, if the file system is writing a 4-KB block to a 16-KB range, it would specify an alignment of 4 KB, thus restricting the write to one of four locations. Without such an option, the disk could logically choose to start writing at any 512-byte offset within the range.

The interface also guarantees no reordering of blocks within a single multi-block write. This decision enables the requester to control the ordering of blocks on disk, which may be important for subsequent read performance, and allows a single target address to be returned by the disk. The interface is summarized in Table 1.

3.1.1 Range Specification

One challenge is to decide how to specify the set of possible target addresses to the disk. The format of this range description must both be compact as well as flexible, which are often at odds.

We initially considered a single range, but found it was too restrictive. For example, a file system may have a large amount of free space on a given track, but the presence of just a few allocated blocks in said track would greatly reduce the utility of single-range range writes. In other words, the client may wish to express that a request can be written anywhere within the range B_{begin} to B_{end} *except* for blocks $B_{begin} + 1$ and $B_{begin} + 2$ (because those blocks are already in use). Thus, the interface needs to support such flexibility.

We also considered a list of target addresses. While this approach is quite flexible, we felt it added too much overhead to each write command. A file system might wish to specify a large number of choices; with a range, in the best case, this is just the start and end of a range; in the list approach, it comprises hundreds or even thousands of target addresses.

Due to these reasons, we believe that there are a few sensible possibilities. One is a simple *list of ranges*; the client specifies a list of begin and end addresses, and the disk is free to write the request within any one such range. A similar effect could be achieved with the combination of a single large range and corresponding *bitmap* which indicates which blocks of the range are free. Both the list-of-ranges and bitmap interfaces are equivalent, as they allow full flexibility in compact forms; we leave the exact specification to the future.

3.1.2 Overlapping Ranges

Modern disks allow multiple outstanding writes. While improving performance, the presence of multiple outstanding requests complicates range writes. In particular, a file system may wish to issue two requests to the disk, R_1 and R_2. Assume both requests should end up near one another on disk (e.g., they are to files that live in the same cylinder group). Assume also that the file system has a free block range in that disk region, B_1 through B_n.

Thus, the file system would like to issue both requests R_1 and R_2 simultaneously, giving each the range B_1 through B_n. However, the disk is thus posed with a problem: how can it ensure it does not write the two requests to the same location?

The simplest solution would be to disallow overlapped writes, much like many of today's disks do not allow multiple outstanding writes to the same address ("overlapped commands" in SCSI parlance [35]). In this case, the file system would have two choices. First, it could serialize the two requests, first issuing R_1, observing which block it was written to (say B_k), and then submitting request

Classic Write
in: `address, data, length`
out: `status`

Range Write
in: **`range descriptor, alignment,`** `data, length`
out: `status,` **`resulting target address`**

Table 1: **Classic vs. Range Writes.** *The table shows the differences between the classic idealized disk write and a range write. The range descriptor can be specified as a list of free ranges or a (begin, end) pair plus bitmap describing the free blocks within the range.*

R_2 with two ranges (B_1 to B_{k-1} and B_{k+1} to B_n). Alternately, the file system could pick subsets of each range (e.g., B_1, B_3, ..., B_{k-1} in one range, and B_2, B_4, ..., B_k in the other), and issue the requests in parallel.

However, the non-overlapped approach was too restrictive; it forces the file system to reduce the number of targets per write request in anticipation of their use and thus reduces performance. Further, non-overlapped ranges complicate the use of range writes, as a client must then make decisions on which portions of the range to give to which requests; this likely decreases the disk's control over low-level placement and thus decreases performance. For these reasons, we chose to allow clients to issue multiple outstanding range writes with overlapping ranges.

Overlapping writes complicate the implementation of range writes within the disk. Consider our example above, where two writes R_1 and R_2 are issued concurrently, each with the range B_1 through B_n. In this example, assume that the disk schedules R_1 first, and places it on block B_1. It is now the responsibility of the disk to ensure that R_2 is written to any block except B_1. Thus, the disk must (temporarily) note that B_1 has been used.

However, this action raises a new question: how long does the disk have to remember the fact that B_1 was written to and thus should not be considered as a possible write target? One might think that the disk could forget this knowledge once it has reported that R_1 has completed (and thus indicated that B_1 was chosen). However, because the file system may be concurrently issuing another request R_3 to the same range, a race condition could ensue and block B_1 could be (erroneously) overwritten.

Thus, we chose to add a new kind of write barrier to the protocol. A file system uses this feature as follows. First, the file system issues a number of outstanding writes, potentially to overlapping ranges. The disk starts servicing them, and in doing so, tracks which blocks in each range are written. At some point, the file system issues a barrier. The barrier guarantees to the disk that all writes following the barrier take into account the allocation decisions of the disk for the writes before the barrier. Thus, once the disk completes the pre-barrier writes, it can safely forget which blocks it wrote to during that time.

3.2 Beyond Writes: Range Reads

It is of course a natural extension to consider whether range reads should also be supported by a disk. Range reads would be useful in a number of contexts: for example, to pick the rotationally-closest block replica [16, 38], or to implement efficient "dummy reads" in semi-preemptible I/O [9].

However, introducing range reads, in particular for improving rotational costs on reads, requires an expanded interface and implementation from the disk. For example, for a block to be replicated properly to reduce rotational costs, it should be written to opposite sides of a track. Thus, the disk should likely support a replica-creating write which tries to position the blocks properly for later reads. In addition, file systems would have to be substantially modified to support tracking of blocks and their copies, a feature which only a few recent file systems support [31]. Given these and other nuances, we leave range reads for future work.

4 Disk Scheduling

In this section, we describe how an in-disk scheduler must evolve to support range writes. We present two approaches. The first we call *expand-and-cancel scheduling*, a technique that is simple, integrates well with existing schedulers, and performs well, but may be too computationally intensive. Because of this weakness, we present a competing approach known as *hierarchical-range scheduling*, which requires a more extensive restructuring of a scheduler to become range aware but thus avoids excessive computational overheads.

Note that we focus on obtaining the highest performance possible, and thus consider variants of shortest-positioning-time-first schedulers (SPTF). Standard modifications could be made to address fairness issues [18, 28].

4.1 Expand-and-cancel Scheduling

Internally, the in-disk scheduler must be modified to support servicing of requests within lists of ranges. One simple way to implement support for range writes would be through a new scheduling approach we call *expand-and-cancel scheduling (ECS)*, as shown in Figure 2. In the expand phase, a range write request R to block range B_1 through B_n is expanded into n independent requests, R_1 through R_n each to a single location, B_1 through B_n, respectively. When the first of these requests complete (as dictated by the policy of the scheduler), the other requests are canceled (i.e., removed from the scheduling queue). Given any scheduler, ECS guarantees that the best scheduling decision over all range writes (and other requests) will be made, to the extent possible given the current scheduling algorithm.

The main advantage of ECS is its excellent integration with existing disk schedulers. The basic scheduling pol-

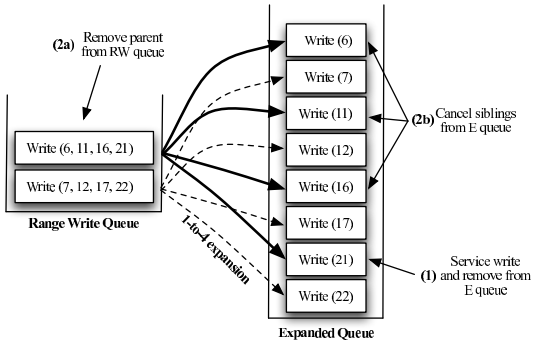


Figure 2: **Expand-and-cancel Scheduling.** *The figure depicts how expand-and-cancel scheduling operates. Range writes are placed into the leftmost queue and then expanded into the full set of writes on the right. In step 0 (not shown), two range writes arrive simultaneously and are placed in the range write queue on the left; their expansions are placed in the expanded queue on the right. In step 1, the scheduler (which examines all requests in the expanded queue and greedily chooses the one with minimal latency) decides to service the write to 21. As a result, the range write to (6, 11, 16, 21) is removed from the range write queue (step 2a), and the expanded requests to 6, 11, and 16 are canceled (step 2b).*

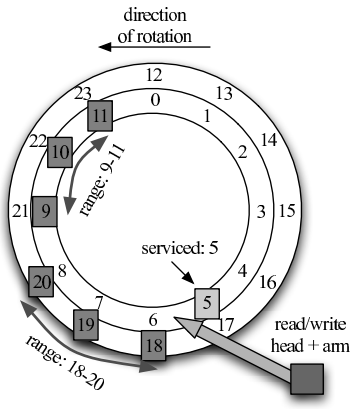


Figure 3: **Hierarchical-range Scheduling.** *The figure depicts how hierarchical-range scheduling works. For the current request, the scheduler must choose which of two possible ranges to write to (18-20 on the adjacent track or 9-11 on the current). The scheduler just serviced a request to 5, and thus must choose whether to stay on the current track and wait for range 9-11 to rotate under the head or switch tracks and write to 18-20. Depending on the relative costs of switching tracks and rotational delay, HRS will decide to which range to write.*

icy is not modified, but simply works with more requests to decide what is the best decision. However, this approach can be computationally expensive, requiring extensive queue reshuffling as range writes enter the system, as well as after the completion of each range write. Further, with large ranges, the size of the expanded queue grows quite large; thus the number of options that must be examined to make a scheduling decision may become computationally prohibitive.

Thus, we expect that disk implementations that choose ECS will vary in exactly how much expansion is performed. By choosing a subset of each range request (e.g., 2 or 4 or 8 target destinations, equally spaced around a track), the disk can keep computational overheads low while still reducing rotational overhead substantially. More expensive drives can include additional processing capabilities to enable more targets, thus allowing for differentiation among drive vendors.

4.2 Hierarchical-Range Scheduling

As an alternative to ECS, we present an approach we call *hierarchical-range scheduling (HRS)*. HRS requires more exact knowledge of drive details, including the current head position, and thus demands a more extensive reworking of the scheduling infrastructure. However, the added complexity comes with a benefit: HRS is much more computationally efficient than ECS, doing less work to obtain similar performance benefits.

HRS works as follows. Given a set of ranges (assuming for now that each range fits within a track), HRS determines the time it takes to seek and settle on the track of each request and the resulting head position. If the head is within the range on that track, HRS chooses the next closest spot within the range as the target, and thus estimates the total latency of positioning (roughly the cost of the seek and settling time). If the head is outside the range, HRS includes an additional rotational cost to get to the first block of the range. Because HRS knows the time these close-by requests will take, it can avoid considering those requests whose seek times already exceed the current best value. In this manner, HRS can consider many fewer options and still minimize rotational costs.

An example of the type of decision HRS makes is found in Figure 3. In the figure, two ranges are available: 9-11 (on the current track) and 18-20 (on an adjacent, outer track). The disk has just serviced a request to block 5 on the inner track, and thus must choose a target for the current write. HRS observes that range 9-11 is on the same track and thus estimates the time to write to 9-11 is the time to wait until 9 rotates under the head. Then, for the 18-20 range, HRS first estimates the time to move the arm to the adjacent track; with this time in hand, HRS can estimate where the head will be relative to the range. If the seek to the outer track is fast, HRS determines that the head will be within the range, and thus chooses the next block in the range as a target. If, however, the short seek takes too long, the arm may arrive and be ready to write just after the range has rotated underneath the head (say at block 21). In this case, HRS estimates the cost of writing to 18-20 as the time to rotate 18 back under the head again, and thus would choose to instead write to block 9 in the first range.

A slight complication arises when a range spans multiple tracks. In this case, for each range, HRS splits the

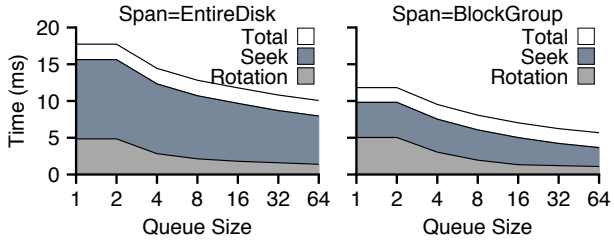


Figure 4: **No Range Writes.** *The figure plots the performance of an in-disk SPTF scheduler on a workload of writes to random locations. The x-axis varies the number of outstanding requests, and the y-axis plots the time per write. The leftmost graph plots performance of writes to random locations over the entire disk; the rightmost graph plots performance of random writes to a 4096-block group.*

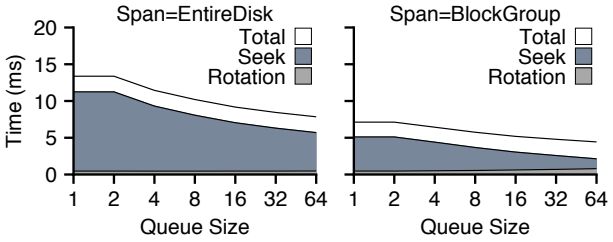


Figure 5: **Track-sized Ranges.** *The graphs plot the performance of range writes under randomized write workloads using the hierarchical range scheduler. The experiments are identical to those described in Figure 4, except that range writes are used instead of traditional writes; the range is set to 100 blocks, just bigger than the track size of the simulated disk (thus eliminating rotation).*

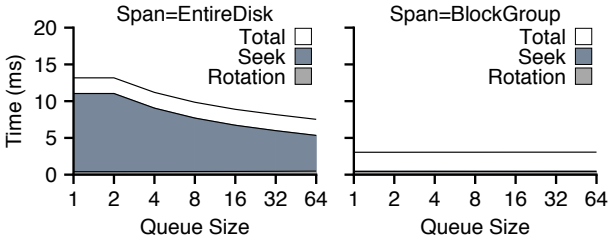


Figure 6: **Group-sized Ranges.** *The graph plots performance of range writes, as described in Figure 5. The small difference: range size is set to 4096 blocks (the size of a block group).*

request into a series of related requests, each of which fit within a single track. Then, HRS considers each in turn as before. Lists of ranges are similarly handled.

4.3 Methodology

We now wish to evaluate the utility of range writes in disk schedulers. To do so, we utilize a detailed simulation environment built within the DiskSim framework [5].

We made numerous small changes throughout DiskSim to provide support for range writes. We implemented a small change to the interface so pass range descriptors to the disk, and more extensive changes to the SPTF scheduler to implement both EC and HR scheduling. Overall, we changed or added roughly one thousand lines of code to the simulator. Unless explicitly investigating EC scheduling, we use the HR scheduler.

For all simulated experiments, we use the HP C2247A disk, which has a rotational speed of 5400 RPM, and a relatively small track size (roughly 60-100 blocks, depending on the zone). It is an older model, and thus, as compared to modern disks, its absolute positioning costs are high whereas track size and data transfer rates are low. However, when writing to a localized portion of disk, the relative balance between seek and rotation is reasonable; thus we believe our results on reductions in positioning time should generalize to modern disks.

4.4 Experiments

4.4.1 Do multiple outstanding requests solve the positioning-time problem?

Traditional systems attack the positioning-time problem by sending multiple requests to the disk at once; internally, an SPTF scheduler can reorder said requests and reduce positioning costs [28]. Thus, the first question we address is whether the presence of multiple outstanding requests solves the positioning-time problem.

In this set of experiments, we vary the number of outstanding requests to the disk under a randomized write workload and utilize an in-disk SPTF scheduler. Each experiment varies the *span* of the workload; a span of the entire disk implies the target address for the write was chosen at random from the disk; a span of a block group implies the write was chosen from a localized portion of the disk (from 4096 blocks, akin to a FFS cylinder group). Figure 4 presents our results.

From the graphs, we make three observations. First, at the left of each graph (with only 1 or 2 outstanding requests), we see the large amount of time spent in seek and rotation. Range writes will be of particular help here, potentially reducing request times dramatically. Second, from the right side of each graph (with 64 outstanding requests), we observe that positioning times have been substantially reduced, but not removed altogether. Thus, flexibility in exact write location as provided by range writes could help reduce these costs even further. Third, we see that in comparing the graphs, the span greatly impacts seek times; workloads with locality reduce seek costs while still incurring a rotational penalty.

We also note that having a queue depth of two is no better than having a queue depth of one. Two outstanding requests does not improve performance because in the steady state, the scheduler is servicing one request while another is being sent to the disk, thus removing the possibility of choosing the "better" request.

4.4.2 What is the benefit of range writes?

We now wish to investigate how range writes can improve performance. Figures 5 and 6 presents the results of a set of experiments that use range writes with a small (track-

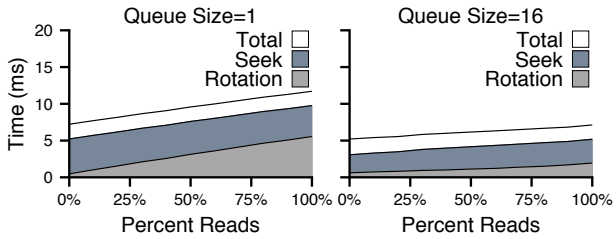


Figure 7: **Mixing in Reads.** *The graphs plot the performance of range writes to random destinations when there is some percentage of reads mixed in. We utilize track-sized ranges and write randomly to locations within a block group. The x-axis varies the percent of reads from 0% to 100%, and the y-axis plots the average time per operation (read or write). Finally, the graph on the left has 1 outstanding request to disk, whereas the graph on the right has 16.*

sized) or large (block-group-sized) amount of flexibility. We again perform random writes to either the entire disk or to a single block group.

Figure 5 shows how a small amount of flexibility can greatly improve performance. By specifying track-sized ranges, all rotational costs are eliminated, leaving only seek overhead and transfer time. We can also see that track-sized range writes are most effective when there are few outstanding requests (the left side of each graph); when the span is set to a block group, for example, positioning costs are halved. Finally, we can also see from this graph that range writes are still of benefit with medium-to-large disk queues, but the benefit is indeed smaller.

Figure 6 plots the results of the same experiment, however with more flexibility: range size is now set to the entire block group. When the span of the experiment is the entire disk (left graph), this makes little difference; rotational delay is eliminated. However, the right graph with a span of a block group shows how range writes can also reduce seek costs. Each write in this experiment can be directed to any free spot in the block group; the result is that there is essentially no positioning overhead, and almost all time spent in transfer.

4.4.3 What if there are reads in the workload?

We have now seen the benefits of range writes in synthetic workloads consisting purely of writes. We now include reads in the workload, and show the diminishing benefit of range writes in read-dominated workloads. Figure 7 plots the results of our experiments.

From the figures, we observe the expected result that with an increasing percentage of reads, the benefit of range writes diminishes. However, we can also see that for many interesting points in the read/write mix, range writes could be useful. With a small number of outstanding requests to the disk and a reasonable percentage of writes, range writes decrease positioning time noticeably.

4.4.4 What is the difference between ECS and HRS?

We next analyze the costs of EC scheduling and HR scheduling. Most of the work that is done by either is the estimation of service time for each possible candidate request; thus, we compare the number of such estimations to gain insight on the computational costs of each approach.

Assume that the size of a range is S, and the size of a track on the disk is T. Also assume that the disk supports Q outstanding requests at a given time (i.e., the queue size). We can thus derive the amount of work that needs to be performed by each approach. For simplicity, we assume each request is a write of a single block (generalizing to larger block sizes is straightforward).

For EC scheduling, assuming the full expansion, each single request in the range-write queue expands into S requests in the expanded queue. Thus, the amount of work, W, performed by ECS is:

$$W_{EC} = S \cdot Q \tag{1}$$

In contrast, HR scheduling takes each range and divides it into a set of requests, each of which is contained within a track. Thus, the amount of work performed by HRS is:

$$W_{HR} = \lceil \frac{S}{T} \rceil \cdot Q \tag{2}$$

However, HRS need not consider all these possibilities. Specifically, once the seek time to a track is higher than the current best seek plus rotate, HRS can stop considering whether to schedule this and other requests that are on tracks that are further away. The worst case number of tracks that must be considered is thus bounded by the number of tracks one can reach within the time of a revolution plus the time to seek to the nearest track. Thus, the equation above represents an upper bound on the amount of work HRS will do.

Even so, the equations make clear why HR scheduling performs much less work than EC scheduling in most cases. For example, assuming that a file system issues range writes that roughly match track size ($S = T$), the amount of work performed by HRS is roughly Q. In contrast, ECS still performs $S \cdot Q$ work; as track sizes can be in the thousands, ECS will have to execute a thousand times more work to achieve equivalent performance.

4.4.5 How many options does ECS need?

Finally, given that EC scheduling cannot consider the full range of options, we investigate how many options such a scheduler requires to obtain good performance. To answer this question, we present a simple workload which repeatedly writes to the same track, and vary the number of target options it is given. Figure 8 presents the results.

In this experiment, we assume that if there exists only a single option, it is to the same block of the track; thus, successive writes to the same block incur a full rotational

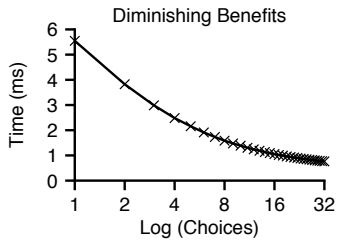


Figure 8: **The Diminishing Benefits of More Choice.** *The figure plots the performance of successive write requests to the same track. Along the x-axis, we increase the number of choices available for write targets, and the y-axis plots average write time.*

delay. As further options are made available to the scheduler, they are equally spaced around the track, maximizing their performance benefit.

From this figure, we can conclude that ECS does not necessarily need to consider all possible options within a range to achieve most of the performance benefit, as expected. By expanding a entire-track range to just eight choices that are properly spaced out across the track, most of the performance benefits can be achieved.

However, this example simplifies that problem quite a bit. For ranges that are larger than a single track, the expansion becomes more challenging; exactly how this should be done remains an open problem.

4.5 Summary

Our study of disk scheduling has revealed a number of interesting results. First, the overhead of positioning time cannot be avoided with traditional SPTF scheduling alone; even with multiple outstanding requests to the disk, seek and rotational overheads still exist.

Second, range writes can dramatically improve performance relative to SPTF scheduling, reducing both rotational and seek costs. To achieve the best performance, a file system (or other client) should give reasonably large ranges to the disk: track-sized ranges remove rotational costs, while larger ranges help to noticeably reduce seek time. Although range writes are of greatest utility when there are only a few outstanding writes to the disk, range writes are still useful when there are many.

Third, the presence of reads in a workload obviously reduces the overall effect of range writes. However, range writes can have a noticeable impact even in relatively balanced settings.

Finally, both the EC and HR schedulers perform well, and thus are possible candidates for use within a disk that supports range writes. If one is willing to rewrite the scheduler, HR is the best candidate. However, if one wishes to use the simpler EC approach, one must do so carefully: the full expansion of ranges exacts a high computational overhead.

5 Integrating Range Writes into Classic File System Allocation

In this section, we explore the issues a file system must address in order to incorporate range writes into its allocation policies. We first discuss the issues in a general setting, and then describe our effort in building a detailed ext2 file system simulator to explore how these issues can be tackled in the context of an existing system.

5.1 File System Issues

There are numerous considerations a file system must take into account when utilizing range writes. Some complicate the file system code, some have performance ramifications, and some aspects of current file systems simply make using range writes difficult or impossible. We discuss these issues here.

5.1.1 Preserving Sequentiality

One problem faced by file systems utilizing range writes is the loss of exact control over placement of files. However, as most file systems only have approximate placement as their main goal (e.g., allocate a file in the same group as its inode), loss of detailed control is acceptable.

Loss of sequentiality, however, would present a larger problem. For example, if a file system freely writes blocks of a file to non-consecutive disk locations, reading back the file would suffer inordinately poor performance. To avoid this problem, the file system should present the disk with larger writes (which the disk will guarantee are kept together), or restrict ranges of writes to make it quite likely that the file will end up in sequential or near-sequential order on disk.

5.1.2 Determining Proper Ranges

Another problem that arises for the file system is determining the proper range for a request. How much flexibility is needed by the disk in order to perform well?

In general, the larger the range given to the disk, the more positioning time is reduced. The simulation results presented in Section 4 indicate that track-sized ranges effectively remove rotational costs while larger sized ranges (e.g., several thousand blocks) help with seek costs. In the ideal case, positioning time can be almost entirely removed if the size of the target range matches the span of the current workload.

Thus, the file system should specify the largest range that best matches its allocation and layout policy. For example, FFS could specify that a write be performed to any free block within a cylinder group.

5.1.3 Bookkeeping

One major change required of the file system is how it handles a fundamental problem with range writes which we refer to as *delayed address notification*. Specifically, only as each write completes does the file system know the

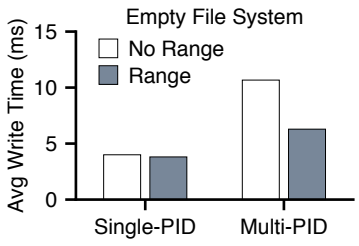


Figure 9: **File Create Time (Empty File System).** *The figure plots the average write time during a file create benchmark. The benchmark creates 1000 4-KB files in the same directory. Range writes are either used or not, and the files are either created by a single process or multiple processes. The y-axis plots the average write time of each write across the 1000 data block writes that occur.*

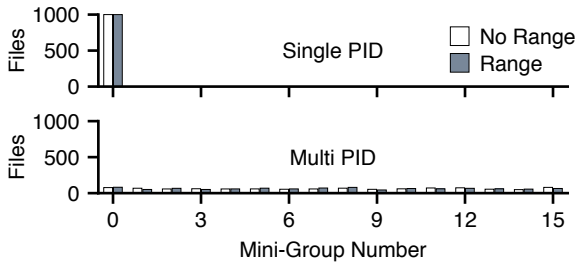


Figure 10: **File Placement.** *The figure shows how files were placed per mini-group across two different experiments. In the first, a single process (PID) created 1000 files; in the second, each file was created by a different PID. The x-axis plots the mini-group number, and the y-axis shows the number of files that were placed in the mini-group, for both range writes and traditional writes.*

target address of the write. The file system cares about this result because it is in charge of bookkeeping, and must record the address in a pertinent structure (e.g., an inode).

In general, this may force two alterations in file system allocation. First, the file system must carefully track outstanding requests to a given region of disk, in order to avoid sending writes to a full region. However, this modification should not be substantial.

Second, delayed notification forces an ordering on file systems, in that the block pointed to must be written before the block containing the pointer. Although reminiscent of soft updates [12], this ordering should be easier to implement, because the file system will likely not employ range writes for all structures, as we discuss below.

5.1.4 Inflexible Structures

Finally, while range writes are quite easy to use for certain block types (e.g., data blocks), other fixed structures are more problematic. For example, consider inodes in a standard FFS-like file system. Each inode shares a block with many others (say 64 per 4-KB block). Writing an inode block to a new location would require the file system to give each inode a new inode number; doing so necessitates finding every directory in the system that contains those inode numbers and updating them.

Thus, we believe that range writes will likely be used at first only for the most flexible of file system structures. Over time, as file systems become more flexible in their placement of structures, range writes can more broadly be applied. Fortunately, modern file systems have more flexible structures; for example, Sun's ZFS [31], NetApp's WAFL [14], and LFS [25] all take a "write-anywhere" approach for most on-disk structures.

5.2 Incorporating Range Writes into ext2

We now present our experience of incorporating range writes into a simulation we built of Linux ext2. Allocation in ext2 (and ext3) derives from classic FFS allocation [22] but has a number of nuances included over the years to improve performance. We now describe the basic allocation policies.

When creating a directory, the "Orlov" allocation algorithm is used. In this algorithm, top-level directories are spread out by searching for the block group with the least number of subdirectories and an above-average free block count and free-inode count. Other directories are placed in a block group meeting a minimum threshold of free inodes and data blocks and having a low directory-to-file ratio. In both cases the parent's block group is preferred given that it meets all criteria.

The allocation of data blocks is done by choosing a goal block and searching for the nearest free block to the goal. For the first data block in the file the goal is found by choosing a block in the same group as the inode. The specific block is chosen by using the PID of the calling process to select one of 16 start locations within the block group; we call each of these 16 locations a *mini-group* within the greater block group. The desire here is to place "functionally related" files closer on disk. All subsequent data block allocations for a given file have the goal set to the next sequential block.

To utilize range writes, our variant of ext2 tries to follow the basic constraints of the existing ext2 policies. For example, if the only constraint is that a file is placed within a block group, than we issue a range write that specifies the free ranges within that group. If the policy wishes to place the file within a mini-group, the range writes issued for that file are similarly constrained. We also make sure to preserve sequentiality of files. Thus, once a file's first block is written to disk, subsequent blocks are placed contiguously beyond it (when possible).

5.3 Methodology

To investigate ext2 use of range writes, we built a detailed file system simulator. The simulator implements all of the policies above (as well as a few others not relevant for this section) and is built on top of DiskSim. The simulator

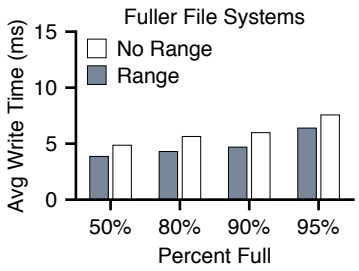


Figure 11: **File Create Time (Fuller File System).** *The figure plots the average write time during a file create benchmark. The benchmark creates 1000 4-KB files in the same directory. Range writes are either used or not, and the files are either created by a single process. The x-axis varies the fullness of the block group.*

	Traditional ext2	with Range Writes
Untar	143.0	123.1
PostMark	29.9	22.2
Andrew	23.2	23.4

Table 2: **File System Workloads.** *Each row plots the performance (in seconds) of a simulated workload. In the left column, results represent the time taken to run the workload on our simulated standard ext2, whereas on the right, the time to run the workload on ext2 with range writes is presented. Three workloads are employed: untar, which unpacks the Linux source tree; PostMark, which emulates the workload of an email server (by creating, accessing, and deleting files), using its default settings; and the modified Andrew benchmark, which emulates typical user behavior. The simulations were driven by file-system-level traces of the given workloads which were then played back against our simulated file system.*

presents a file system API, and takes in a trace file which allows one to exercise the API and thus the file system. The simulator also implements a simple caching infrastructure, and writes to disk happen in a completely unordered and asynchronous fashion (akin to ext2 mounted asynchronously). We use the same simulated disk as before (the HP C2247A), set the disk-queue depth to 16, and utilize HR scheduling.

5.4 Results

5.4.1 Small-File Creation on Empty File Systems

We first show how flexible data block placement can improve performance. For this set of experiments, we simply create a large number of small files in a single directory. Thus, the file system should create these files in a single block group, when there is space. For this experiment, we assume that the block group is empty to start.

Figure 9 shows the performance of small-file allocation both with and without range writes. When coming from a single process, using range writes does not help much, as all file data are created within the same mini-group and indeed are placed contiguously on disk. However, when coming from different processes, we can see the benefits of using range writes. Because these file allocations get spread across multiple mini-groups within the block group, the flexibility of range writes helps reduce seek and rotation time substantially.

We also wish to ensure that our range-aware file system makes similar placement decisions within the confines of the ext2 allocation policies. Thus, Figure 10 presents the breakdowns of which mini-group each file was placed in. As one can see from the figure, the placement decisions of range writes, in both the single-process and multi-process experiments, closely follow that of the traditional ext2. Thus, although the fine-grained control of file placement is governed by the disk, the coarse-grained control of file placement is as desired.

5.4.2 Small-File Creation on Fuller File Systems

We now move to a case where the block group has data in it to begin. This set of experiments varies the fullness of the block group and runs the same small-file creation benchmark (focusing on the single-PID case). Figure 11 plots the results.

From the figure, we can see that by the time a block group is 50% full, range writes improve performance over classic writes by roughly 20%. This improvement stays roughly constant as the block group fills, even as the average write time of both approaches increases. We can also see the effect of fullness on range writes: with fewer options (as the block group fills), it is roughly 70% slower than it was with an empty block group.

5.4.3 Real Workloads

The first two synthetic benchmarks focused on file creation in empty or partially-full file systems, demonstrating some of the benefits of range writes. We now simulate the performance of an application-level workload. Specifically, we focus on three workloads: untar, which unpacks the Linux source tree, PostMark [19], which simulates the workload of an email server, and the modified Andrew Benchmark [15], which emulates typical user behavior. Table 2 presents the results.

We make the following two observations. First, for workloads that have significant write components (untar, PostMark), range writes boost performance (a 16% speedup for untar and roughly 35% for PostMark). Second, for workloads that are less I/O intensive (Andrew), range writes do not make much difference.

5.5 Summary

Integrating range writes into file system allocation has proven promising. As desired, range writes can improve performance during file creation while following the constraints of the higher-level file system policies. As much of write activity is to newly created files [4, 33], we believe our range-write variant of ext2 will be effective in

practice. Further, although limited to data blocks, our approach is useful because traffic is often dominated by data (and not metadata) writes.

Of course, there is much left to explore. For example, partial-file overwrites present an interesting scenario. For best performance, one should issue a range write even for previously allocated data; thus, overwritten data may be allocated to a new location on the disk. Unfortunately, this strategy can potentially destroy the sequentiality of later reads and should be performed with care. We leave this and many other workload scenarios to future work.

6 Case Study: Log Skipping

We now present a case study that employs range writes to improve journal update performance. Specifically, we show how a journaling file system (Linux ext3 in this case) can readily use range writes to more flexibly choose where each log update should be placed on disk. By doing so, a journaling file system can avoid the rotations that occur when performing many synchronous writes to the journal and thus greatly improve performance.

Whereas the previous section employed simulation to study the benefits of range writes, we now utilize a prototype implementation. Doing so presents an innate problem: how do we experiment with range writes in a real system, when no disk (yet) supports range writes? To remedy this dilemma, we develop a software layer, Bark, that emulates a disk with range writes for this specific application. Our approach suggests a method to build acceptance of new technology: first via software prototype (to demonstrate potential) and later via actual hardware (to realize the full benefits).

6.1 Motivation

The primary problem that we address in this section is how to improve the performance of synchronous writes to a log or journal. Thus, it is important to understand the sequence of operations that occur when the log is updated.

A journaling system writes a number of blocks to the log; these writes occur whenever an application explicitly forces the data or after certain timing intervals. First, the system writes a *descriptor block*, containing information about the log entry, and the actual data to the log. After this write, the file system waits for the descriptor blocks and data to reach the disk and then issues a synchronous *commit block* to the log; the file system must wait until the first write completes before issuing the commit block in case a crash occurs.

In an ideal world, since all of the writes to the log are sequential, the writes would achieve sequential bandwidth. Unfortunately, in a traditional journaling system, the writes do not. Because there is a non-zero time elapsed since the previous block was written, and because the disk keeps rotating at a constant speed, the commit block cannot be written immediately. The sectors that need to be written have already passed under the disk head and thus a rotation is incurred to write the commit block.

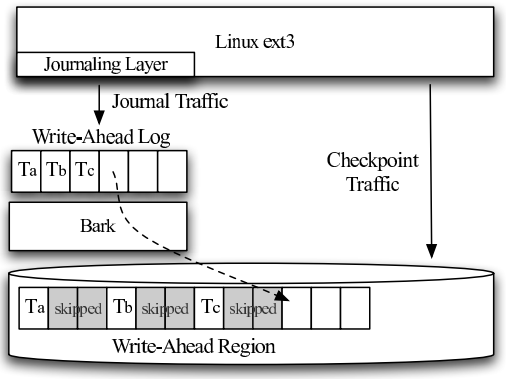


Figure 12: **Bark-itecture.** *The figure illustrates how a file system can be mounted upon Bark to improve journal write performance. All journal traffic is directed through Bark, which picks a skip distance based on think time and the position of the last write to disk. Bark performs this optimization transparently, thus improving the performance of journal writes with no change to the file system above. In the specific example shown, the file system has committed three transactions to disk: Ta, Tb, and Tc. Bark, using its performance model, has spread the transactions across the physical disk, leaving empty spaces (denoted as "skipped") in the write-ahead region.*

Our approach is to transform the write-ahead log of a journaling file system into a more flexible *write-ahead region*. Instead of issuing a transaction to the journal in the location directly following the previous transaction, we instead allow the transaction to be written to the next rotationally-closest location. This has the effect of spreading transactions throughout the region with small distances between them, but improves performance by minimizing rotation.

Our approach derives from previous work in database management systems by Gallagher et al. [11]. Therein, the authors describe a simple dynamic approach that continually adjusts the distance to skip in a log write to reduce rotation. Perhaps due to the brief description of their algorithm, we found it challenging to successfully reproduce their results. Instead, we decided on a different approach, first building a detailed performance model of the log region of the disk and then using that to decide how to best place writes to reduce rotational costs. The details of our approach, described below, are based on our previous work in building the disk mimic [23].

We now discuss how we implement write-ahead regions in our prototype system. The biggest challenge to overcome is the lack of range writes in the disk. We describe our software layer, Bark, which builds a model of the performance contours of the log (hence the name) and uses it to issue writes to the journal so as to reduce rotational overheads. We then describe our experiments with the Linux ext3 journal mounted on top of Bark.

	w/o Bark	w/ Bark	Null
Uncached	50.7	42.1	38.8
Cached	44.2	27.3	25.4

Table 3: **Bark Performance.** *Each row of the table plots the overall performance (in seconds) of TPC-B in three different settings: without Bark, with Bark, and on a "null" journal that reports success for writes without performing disk I/O (the null journal represents the best possible improvement possible by using Bark). The first row reports performance of a cold run, where table reads go to disk. The second row reports performance when the table is in cache (i.e., only writes go to disk). Experiments were run upon a Sun Ultra20 with 1 GB of memory and two Hitachi Deskstar 7K80 drives. The average of three runs is reported; there was little deviation in the results.*

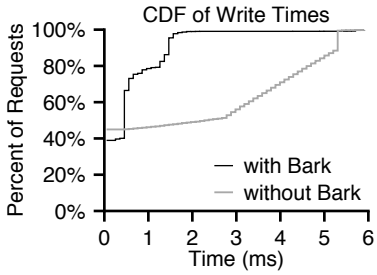


Figure 13: **Write Costs.** *The figure plots the cumulative distribution of write request times during TPC-B. Two lines are plotted: the first shows the cost of writes through Bark, whereas the second shows costs without. The data is taken from a "cached" run as described above.*

6.2 Log-Performance Modeling

Bark is a layer that sits between the file system and disk and redirects journal writes so as to reduce rotational overhead. To do so, Bark builds a performance model of the log *a priori* and uses it to decide where best to write the next log write.

Our approach builds on our previous work that measures the request time between all possible pairs of disk addresses in order to perform disk scheduling [23]. Our problem here is simpler: Bark must simply predict where to place the next write in order to reduce rotation.

To make this prediction, Bark performs measurements of the cost of writes to the portion of the disk of interest, varying both the distance between writes (the "skip" size) and think time between requests. The data is stored in a table and made available to Bark at runtime.

For the results reported in this paper, we created a disk profile by keeping a fixed write size of 4 KB (the size of a block), and varying the think time from 0 ms to 80 ms in intervals of 50 microseconds, and the skip size from 0 KB to 600 KB in intervals of 512 bytes. To gain confidence each experiment was repeated multiple times and the average of the write times was taken.

6.3 From Models to Software

With the performance model in place, we developed Bark as a software pseudo-device that is positioned between the file system journaling code and the disk. Bark thus presents itself to the journaling code as if it were a typical disk of a given size S. Underneath, Bark transparently utilizes more disk space (say $2 \cdot S$) in order to commit journal writes to disk in a rotationally-optimal manner, as dictated by the performance model. Figure 12 depicts this software architecture.

At runtime, Bark receives a write request and must decide exactly where to place it on disk. Given the time elapsed since the last request completed, Bark looks up the required skip distance in the prediction table and uses it to decide where to issue the current write.

Two issues arise in the Bark implementation. The first is the management of free space in the log. Bark keeps a data structure to track which blocks are free in the journal and thus candidates for fast writes. The main challenge for Bark is detecting when a previously-used block becomes free. Bark achieves this by monitoring overwrites by the journaling layer; when a block is overwritten in the logical journal, Bark frees the corresponding physical block to which it had been mapped.

The second issue is support for recovery. Journals are not write-only devices. In particular, during recovery, the file system reads pending transactions from the journal in order to replay them to the file system proper and thus recover the file system to a consistent state. To enable this recovery without file system modification, Bark would need to record a small bit of extra information with each set of contiguous writes, specifically the address in the logical address space to which this write was destined. Doing so would enable Bark to scan the write-ahead region during recovery and reconstruct the logical address space, and thus allows recovery to proceed without any change to the file system code. However, we have not yet fully implemented this feature (early experience suggests it will be straightforward).

6.4 Results

We now measure the performance of unmodified Linux ext3 running on top of Bark. For this set of experiments, we mount the ext3 journal on Bark and let all other checkpoint traffic go to disk directly.

For a workload, we wished to find an application that stressed journal write performance. Thus, we chose to run an implementation of the classic transactional benchmark TPC-B. TPC-B performs a series of debits and credits to a simple set of database tables. Because TPC-B forces data to disk frequently, it induces a great deal of synchronous I/O traffic to the ext3 journal.

Table 3 plots the performance of TPC-B on Linux ext3 in three separate scenarios. In the first, the unmodified traditional journaling approach is used; in the second, Bark is used underneath the journal; in the third, we implement

a fast "null" journal which simply returns success when given a write without doing any work. This last option serves as an upper-bound on performance improvements realized through more efficient journaling.

Note also that each row varies whether table reads go to disk (uncached) or are found in memory (cached). In the cached runs, most table reads hit in memory (and thus disk traffic is dominated by writes). By measuring performance in the uncached scenario, we can determine the utility of our approach in scenarios where there are reads present in the workload; the cached workload stresses write performance and thus presents a best-case for Bark under TPC-B.

From the graph, we can see that Bark greatly improves the overall runtime of TPC-B; Bark achieves a 20% speedup in the uncached case and over 61% in the cached run. Both of these approach the optimal time as measured by the "null" case. Thus, beyond the simulation results presented in previous sections, Bark shows that range writes can work well in the real world as well.

Figure 13 sheds some light on this improvement in performance. Therein we plot the cumulative distribution of journal-write times across all requests during the cached run. When using Bark, most journal writes complete quickly, as they have been rotationally well placed through our simple skipping mechanism. In contrast, writes to the journal without Bark take much longer on average, and are spread across the rotational spectrum of the disk drive.

6.5 Discussion

We learned a number of important lessons from our implementation of log skipping using range writes. First, we see that range writes are also useful for a file system journal. Under certain workloads, journaling can induce a large rotational cost; freedom to place transactions to a free spot in the journal can greatly improve performance.

Second, with read traffic present, the improvement seen by Bark is lessened but still quite noticeable. Thus, even with reads (in the uncached case, they comprise roughly one-third of the traffic to the main file system), flexible writes to the journal improve performance.

Finally, we should note that we chose to incorporate flexible writes underneath the file system in the simplest possible way, without changing the file system implementation at all. If range writes actually existed within the disk, the Bark layer would be much simpler: it would issue the range writes to disk instead of using a model to find the next fast location to write to. A different approach would be to modify the file system code and change the journaling layer to support range writes directly, something we plan to do in future work.

7 Conclusions

We have presented a small but important change to the storage interface, known as range writes. By allowing the file system to express flexibility in the exact write location, the disk is free to make better decisions for write targets and thus improve performance.

We believe that the key element of range writes is their evolutionary nature; there is a clear path from the disk of today without range writes to the disk of tomorrow with them. This fact is crucial for established industries, where change is fraught with many complications, both practical and technical; for example, consider object-based drives, which have taken roughly a decade to begin to come to market [13].

Interestingly, the world of storage may be in the midst of a revolution as solid-state devices become more of a marketplace reality. Fortunately, we believe that range writes are still quite useful in this and other new environments. By letting the storage system take responsibility for low-level placement decisions, range writes enable high performance through device-specific optimizations. Further, range writes naturally support functionality such as wear-leveling, and thus may also help increase device lifetime while reducing internal complexity.

We believe there are numerous interesting future paths for range writes, as we have alluded to throughout the paper. The corollary operation, range reads, presents new challenges but may realize new benefits. Integration into RAID systems introduces intriguing problems as well; for example, parity-based schemes often assume a fixed offset placement of blocks within a stripe across drives. An elegant approach to adding range writes into RAIDs may well pave the way for acceptance of this technology into the higher end of the storage system arena.

Finding the right interface between two systems is always challenging. Too much change, and there will be no adoption; too little change, and there is no significant benefit. We believe range writes present a happy medium: a small interface change with large performance gains.

Acknowledgments

We thank the members of our research group for their insightful comments. We would also like to thank our shepherd Phil Levis and the anonymous reviewers for their excellent feedback and comments, all of which helped to greatly improve this paper.

This work is supported by the National Science Foundation under the following grants: CCF-0621487, CNS-0509474, CCR-0133456, as well as by generous donations from Network Appliance and Sun Microsystems.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] Dave Anderson. OSD Drives. www.snia.org/events/past/developer2005/0507_v1_DBA_SNIA_OSD.pdf, 2005.

[2] Dave Anderson, Jim Dykes, and Erik Riedel. More Than an Interface: SCSI vs. ATA. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, California, April 2003.

[3] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, Pacific Grove, California, October 1991.

[4] Mary Baker, John Hartman, Martin Kupfer, Ken Shirriff, and John Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 198–212, Pacific Grove, California, October 1991.

[5] John S. Bucy and Gregory R. Ganger. The DiskSim Simulation Environment Version 3.0 Reference Manual. Technical Report CMU-CS-03-102, Carnegie Mellon University, January 2003.

[6] Harry Chambers. *My Way or the Highway: The Micromanagement Survival Guide*. Berrett-Koehler Publishers, 2004.

[7] Chia Chao, Robert English, David Jacobson, Alexander Stepanov, and John Wilkes. Mime: a high performance parallel storage device with strong recovery guarantees. Technical Report HPL-CSP-92-9rev1, HP Laboratories, November 1992.

[8] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The Logical Disk: A New Approach to Improving File Systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 15–28, Asheville, North Carolina, December 1993.

[9] Zoran Dimitrijevic, Raju Rangaswami, and Edward Chang. Design and Implementation of Semi-preemptible IO. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 145–158, San Francisco, California, April 2003.

[10] Robert M. English and Alexander A. Stepanov. Loge: A Self-Organizing Disk Controller. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '92)*, pages 237–252, San Francisco, California, January 1992.

[11] Bill Gallagher, Dean Jacobs, and Anno Langen. A High-performance, Transactional Filestore for Application Servers. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*, pages 868–872, Baltimore, Maryland, June 2005.

[12] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 49–60, Monterey, California, November 1994.

[13] Garth A. Gibson, David Rochberg, Jim Zelenka, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, and Erik Riedel. File server scaling with network-attached secure disks. In *Proceedings of the 1997 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/PERFORMANCE '97)*, pages 272–284, Seattle, Washington, June 1997.

[14] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.

[15] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[16] Hai Huang, Wanda Hung, and Kang G. Shin. FS2: dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 263–276, Brighton, United Kingdom, October 2005.

[17] L. Huang and T. Chiueh. Implementation of a Rotation-Latency-Sensitive Disk Scheduler. Technical Report ECSL-TR81, SUNY, Stony Brook, March 2000.

[18] D. M. Jacobson and J. Wilkes. Disk Scheduling Algorithms Based on Rotational Position. Technical Report HPL-CSP-91-7, Hewlett Packard Laboratories, 1991.

[19] Jeffrey Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., October 1997.

[20] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Summer. One-level Storage System. *IRE Transactions on Electronic Computers*, EC-11:223–235, April 1962.

[21] Charles M. Kozierok. Overview and History of the SCSI Interface. http://www.pcguide.com/ref/hdd/if/scsi/overc.html, 2001.

[22] Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[23] Florentina I. Popovici, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Robust, Portable I/O Scheduling with the Disk Mimic. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, pages 297–310, San Antonio, Texas, June 2003.

[24] Peter M. Ridge and Gary Field. *The Book of SCSI 2/E*. No Starch, June 2000.

[25] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[26] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[27] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, Monterey, California, January 2002.

[28] Margo Seltzer, Peter Chen, and John Ousterhout. Disk Scheduling Revisited. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '90)*, pages 313–324, Washington, D.C, January 1990.

[29] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File System Logging versus Clustering: A Performance Comparison. In *Proceedings of the USENIX Annual Technical Conference (USENIX '95)*, pages 249–264, New Orleans, Louisiana, January 1995.

[30] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Life or Death at Block Level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 379–394, San Francisco, California, December 2004.

[31] Sun Microsystems. ZFS: The last word in file systems. www.sun.com/2004-0914/feature/, 2006.

[32] Nisha Talagala, Remzi H. Arpaci-Dusseau, and Dave Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. Technical Report CSD-99-1063, University of California, Berkeley, 1999.

[33] Werner Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 93–109, Kiawah Island Resort, South Carolina, December 1999.

[34] Randy Wang, Thomas E. Anderson, and David A. Patterson. Virtual Log-Based File Systems for a Programmable Disk. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, Louisiana, February 1999.

[35] Ralph O. Weber. SCSI Architecture Model - 3 (SAM-3). http://www.t10.org/ftp/t10/drafts/sam3/sam3r14.pdf, September 2004.

[36] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling Algorithms for Modern Disk Drives. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '94)*, pages 241–251, Nashville, Tennessee, May 1994.

[37] Bruce L. Worthington, Greg R. Ganger, Yale N. Patt, and John Wilkes. On-Line Extraction of SCSI Disk Drive Parameters. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '95)*, pages 146–156, Ottawa, Canada, May 1995.

[38] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading Capacity for Performance in a Disk Array. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, California, October 2000.

Binary Translation Using Peephole Superoptimizers

Sorav Bansal
Computer Systems Lab
Stanford University
sbansal@cs.stanford.edu

Alex Aiken
Computer Systems Lab
Stanford University
aiken@cs.stanford.edu

Abstract

We present a new scheme for performing binary translation that produces code comparable to or better than existing binary translators with much less engineering effort. Instead of hand-coding the translation from one instruction set to another, our approach automatically learns translation rules using superoptimization techniques. We have implemented a PowerPC-x86 binary translator and report results on small and large compute-intensive benchmarks. When compared to the native compiler, our translated code achieves median performance of 67% on large benchmarks and in some small stress tests actually outperforms the native compiler. We also report comparisons with the open source binary translator Qemu and a commercial tool, Apple's Rosetta. We consistently outperform the former and are comparable to or faster than the latter on all but one benchmark.

1 Introduction

A common worry for machine architects is how to run existing software on new architectures. One way to deal with the problem of software portability is through *binary translation*. Binary translation enables code written for a *source architecture* (or instruction set) to run on another *destination architecture*, without access to the original source code. A good example of the application of binary translation to solve a pressing software portability problem is Apple's Rosetta, which enabled Apple to (almost) transparently move its existing software for the Power-based Macs to a new generation of Intel x86-based computers [1].

Building a good binary translator is not easy, and few good binary translation tools exist today. There are four main difficulties:

1. Some performance is normally lost in translation. Better translators lose less, but even good translators often lose one-third or more of source architecture performance for compute-intensive applications.
2. Because the instruction sets of modern machines tend to be large and idiosyncratic, just writing the translation rules from one architecture to another is a significant engineering challenge, especially if there are significant differences in the semantics of the two instruction sets. This problem is exacerbated by the need to perform optimizations wherever possible to minimize problem (1).
3. Because high-performance translations must exploit architecture-specific semantics to maximize performance, it is challenging to design a binary translator that can be quickly retargeted to new architectures. One popular approach is to design a common intermediate language that covers all source and destination architectures of interest, but to support needed performance this common language generally must be large and complex.
4. If the source and destination architectures have different operating systems then source system calls must be emulated on the destination architecture. Operating systems' large and complex interfaces combined with subtle and sometimes undocumented semantics and bugs make this a major engineering task in itself.

Our work presents a new approach to addressing problems (1)-(3) (we do not address problem (4)). The main idea is that much of the complexity of writing an aggressively optimizing translator between two instruction sets can be eliminated altogether by developing a system that automatically and systematically learns translations. In Section 6 we present performance results showing that this approach is capable of producing destination machine code that is at least competitive with existing state-of-the-art binary translators, addressing problem (1). While we cannot meaningfully compare the en-

gineering effort needed to develop our research project with what goes into commercial tools, we hope to convince the reader that, on its face, automatically learning translations must require far less effort than hand coding translations between architectures, addressing problem (2). Similarly, we believe our approach helps resolve the tension between performance and retargetability: adding a new architecture requires only a parser for the binary format and a description of the instruction set semantics (see Section 3). This is the minimum that any binary translator would require to incorporate a new architecture; in particular, our approach has no intermediate language that must be expanded or tweaked to accommodate the unique features of an additional architecture.

Our system uses *peephole rules* to translate code from one architecture to another. Peephole rules are pattern matching rules that replace one sequence of instructions by another equivalent sequence of instructions. Peephole rules have traditionally been used for compiler-optimizations, where the rules are used to replace a sub-optimal instruction sequence in the code by another equivalent, but faster, sequence. For our binary translator, we use peephole rules that replace a source-architecture instruction sequence by an equivalent destination architecture instruction sequence. For example,

```
ld [r2]; addi 1; st [r2] =>
inc [er3]  { r2 = er3 }
```

is a peephole translation rule from a certain accumulator-based RISC architecture to another CISC architecture. In this case, the rule expresses that the operation of loading a value from memory location `[r2]`, adding 1 to it and storing it back to `[r2]` on the RISC machine can be achieved by a single in-memory increment instruction on location `[er3]` on the CISC machine, where RISC register `r2` is emulated by CISC register `er3`.

The number of peephole rules required to correctly translate a complete executable for any source-destination architecture pair can be huge and manually impossible to write. We automatically learn peephole translation rules using *superoptimization* techniques: essentially, we exhaustively enumerate possible rules and use formal verification techniques to decide whether a candidate rule is a correct translation or not. This process is slow; in our experiments it required about a processor-week to learn enough rules to translate full applications. However, the search for translation rules is only done once, off-line, to construct a binary translator; once discovered, peephole rules are applied to any program using simple pattern matching, as in a standard peephole optimizer. Superoptimization has been previously used in compiler optimization [5, 10, 14], but our work is the first to develop superoptimization techniques for binary translation.

Binary translation preserves execution semantics on two different machines: whatever result is computed on one machine should be computed on the other. More precisely, if the source and destination machines begin in equivalent states and execute the original and translated programs respectively, then they should end in equivalent states. Here, *equivalent states* implies we have a mapping telling us how the states of the two machines are related. In particular, we must decide which registers/memory locations on the destination machine emulate which registers/memory locations of the source machine.

Note that the example peephole translation rule given above is conditioned by the *register map* `r2 = er3`. Only when we have decided on a register map can we compute possible translations. The choice of register map turns out to be a key technical problem: better decisions about the register map (e.g., different choices of destination machine registers to emulate source machine registers) lead to better performing translations. Of course, the choice of instructions to use in the translation also affects the best choice of register map (by, for example, using more or fewer registers), so the two problems are mutually recursive. We present an effective dynamic programming technique that finds the best register map and translation for a given region of code (Section 3.3).

We have implemented a prototype binary translator from PowerPC to x86. Our prototype handles nearly all of the PowerPC and x86 opcodes and using it we have successfully translated large executables and libraries. We report experimental results on a number of small compute-intensive microbenchmarks, where our translator surprisingly often outperforms the native compiler. We also report results on many of the SPEC integer benchmarks, where the translator achieves a median performance of around 67% of natively compiled code and compares favorably with both Qemu [17], an open source binary translator, and Apple's Rosetta [1]. While we believe these results show the usefulness of using superoptimization as a binary translation and optimization tool, there are two caveats to our experiments that we discuss in more detail in Section 6. First, we have not implemented translations of all system calls. As discussed above under problem (4), this is a separate and quite significant engineering issue. We do not believe there is any systematic bias in our results as a result of implementing only enough system calls to run many, but not all, of the SPEC integer benchmarks. Second, our system is currently a static binary translator, while the systems we compare to are dynamic binary translators, which may give our system an advantage in our experiments as time spent in translation is not counted as part of the execution time. There is nothing that prevents our techniques from being used in a dynamic translator; a static translator was

just easier to develop given our initial tool base. We give a detailed analysis of translation time, which allows us to bound the additional cost that would be incurred in a dynamic translator.

In summary, our aim in this work is to demonstrate the ability to develop binary translators with competitive performance at much lower cost. Towards this end, we make the following contributions:

- We present a design for automatically learning binary translations using an off-line search of the space of candidate translation rules.
- We identify the problem of selecting a register map and give an algorithm for simultaneously computing the best register map and translation for a region of code.
- We give experimental results for a prototype PowerPC to x86 translator, which produces consistently high performing translations.

The rest of this paper is organized as follows. We begin with a discussion on the recent applications of binary translation (Section 2). We then provide a brief overview of peephole superoptimizers followed by a discussion on how we employ them for binary translation (Section 3). We discuss other relevant issues involved in binary translation (Section 4) and go on to discuss our prototype implementation (Section 5). We then present our experimental results (Section 6), discuss related work (Section 7), and finally conclude (Section 8).

2 Applications

Before describing our binary translation system, we give a brief overview of a range of applications for binary translation. Traditionally, binary translation has been used to emulate legacy architectures on recent machines. With improved performance, it is now also seen as an acceptable portability solution.

Binary translation is useful to hardware designers for ensuring software availability for their new architectures. While the design and production of new architecture chips complete within a few years, it can take a long time for software to be available on the new machines. To deal with this situation and ensure early adoption of their new designs, computer architects often turn to software solutions like virtual machines and binary translation [7].

Another interesting application of binary translation for hardware vendors is backward and forward compatibility of their architecture generations. To run software written for older generations, newer generations are forced to support backward compatibility. On the flip side, it is often not possible to run newer generation software on older machines. Both of these problems create compatibility headaches for computer architects and huge management overheads for software developers. It is not hard to imagine the use of a good binary-translation based solution to solve both problems in the future.

Binary translation is also being used for machine and application virtualization. Leading virtualization companies are now considering support for allowing the execution of virtual machines from multiple architectures on a single host architecture [20]. Hardware vendors are also developing virtualization platforms that allow people to run popular applications written for other architectures on their machines [16]. Server farms and data centers can use binary translation to consolidate their servers, thus cutting their power and management costs.

People have also used binary translation to improve performance and reduce power consumption in hardware. Transmeta Crusoe [12] employs on-the-fly binary translation to execute x86 instructions on a VLIW architecture thereby cutting power costs [11]. Similarly, in software, many Java virtual machines perform on-the-fly translation from Java bytecode to the host machine instructions [25] to improve execution performance.

3 Binary Translation Using Peephole Superoptimizers

In this section we give a necessarily brief overview of the design and functionality of peephole superoptimizers, focusing on the aspects that are important in the adaptation to binary translation.

3.1 Peephole Superoptimizers

Peephole superoptimizers are an unusual type of compiler optimizer [5, 10, 14], and for brevity we usually refer to a peephole superoptimizer as simply an optimizer. For our purposes, constructing a peephole superoptimizers has three phases:

1. A module called the *harvester* extracts *target instruction sequences* from a set of training programs. These are the instruction sequences we seek to optimize.
2. A module called the *enumerator* enumerates all possible instruction sequences up to a certain length. Each enumerated instruction sequence s is checked to see if it is equivalent to any target instruction sequence t. If s is equivalent to some target sequence t and s is cheaper according to a cost function (e.g., estimated execution time or code size) than any other sequence known to be equivalent to t (including t itself), then s is recorded as the

best known replacement for t. A few sample peephole optimization rules are shown in Table 1.

3. The learned (target sequence, optimal sequence) pairs are organized into a lookup table indexed by target instruction sequence.

Once constructed, the optimizer is applied to an executable by simply looking up target sequences in the executable for a known better replacement. The purpose of using harvested instruction sequences is to focus the search for optimizations on the code sequences (usually generated by other compilers) that appear in actual programs. Typically, all instruction sequences up to length 5 or 6 are harvested, and the enumerator tries all instruction sequences up to length 3 or 4. Even at these lengths, there are billions of enumerated instruction sequences to consider, and techniques for pruning the search space are very important [5]. Thus, the construction of the peephole optimizer is time-consuming, requiring a few processor-days. In contrast, actually applying the peephole optimizations to a program typically completes within a few seconds.

The enumerator's equivalence test is performed in two stages: a fast execution test and a slower boolean test. The execution test is implemented by executing the target sequence and the enumerated sequence on hardware and comparing their outputs on random inputs. If the execution test does not prove that the two sequences are different (i.e., because they produce different outputs on some tested input), the boolean test is used. The equivalence of the two instruction sequences is expressed as boolean formula: each bit of machine state touched by either sequence is encoded as a boolean variable, and the semantics of instructions is encoded using standard logical connectives. A SAT solver is then used to test the formula for satisfiability, which decides whether the two sequences are equal.

Using these techniques, *all* length-3 x86 instruction sequences have previously been enumerated on a single processor in less than two days [5]. This particular superoptimizer is capable of handling opcodes involving flag operations, memory accesses and branches, which on most architectures covers almost all opcodes. Equivalence of instruction sequences involving memory accesses is correctly computed by accounting for the possibility of aliasing. The optimizer also takes into account live register information, allowing it to find many more optimizations because correctness only requires that optimizations preserve live registers (note the live register information qualifying the peephole rules in Table 1).

Target Sequence	Live Registers	Equivalent Enumerated Sequence
`movl (%eax), %ecx` `movl %ecx, (%eax)`	`eax,ecx`	`movl (%eax), %ecx`

`sub %eax, %ecx`
`mov %ecx, %eax`
`dec %eax`
	`eax`	`not %eax`
		`add %ecx, %eax`

`sub %eax, %ecx`
`test %ecx, %ecx`
`je .END`
`mov %edx, %ebx`
`.END:`
	`eax,ecx,`
	`edx,ebx`
		`sub %eax, %ecx`
		`cmovne %edx, %ebx`

Table 1: Examples of peephole rules generated by a superoptimizer for x86 executables

.

3.2 Binary Translation

We discuss how we use a peephole superoptimizer to perform efficient binary translation. The approach is similar to that discussed in Section 3.1, except that now our target sequences belong to the source architecture while the enumerated sequences belong to the destination architecture.

The binary translator's harvester first extracts target sequences from a training set of source-architecture applications. The enumerator then enumerates instruction sequences on the destination architecture checking them for equivalence with any of the target sequences. A key issue is that the definition of equivalence must change in this new setting with different machine architectures. Now, equivalence is meaningful only with respect to a *register map* showing which memory locations on the destination machine, and in particular registers, emulate which memory locations on the source machine; some register maps are shown in Table 2. A register in the source architecture could be mapped to a register or a memory location in the destination architecture. It is also possible for a memory location in the source architecture to be mapped to a register in the destination architecture.

A potential problem is that for a given source-architecture instruction sequence there may be many valid register maps, yielding a large number of (renamed) instruction sequences on the target-architecture that must be tested for equivalence. For example, the two registers used by the PowerPC register move instruction `mr r1,r2` can be mapped to the 8 registers of the x86 in 8*7=56 different ways. Similarly, there may be many variants of a source-architecture instruction sequence. For example, on the 32 register PowerPC, there are 32*31=992 variants of `mr r1, r2`. We avoid these problems by eliminating source-architecture sequences

Register Map	Description
r1→eax	Maps PowerPC register to x86 register
r1→M_1	Maps PowerPC register to a memory location
M_s→eax	Maps a memory location in source code to a register in the translated code
r1→eax r2→eax	Invalid. Cannot map two PowerPC registers to the same x86 register
M_s→M_t	Maps one memory location to another (e.g. address space translation)

Table 2: Some valid (and invalid) register maps from PowerPC-x86 translation (M_i refers to a memory location).

that are register renamings of one canonically-named sequence and by considering only one register map for all register maps that are register renamings of each other. During translation, a target sequence is (canonically) renamed before searching for a match in the peephole table and the resulting translated sequence is renamed back before writing it out. Further details of this *canonicalization* optimization are in [5].

When an enumerated sequence is found to be equivalent to a target sequence, the corresponding peephole rule is added to the translation table together with the register map under which the translation is valid. Some examples of peephole translation rules are shown in Table 3.

Once the binary translator is constructed, using it is relatively simple. The translation rules are applied to the source-architecture code to obtain destination-architecture code. The application of translation rules is more involved than the application of optimization rules. Now, we also need to select the register map for each code point before generating the corresponding translated code. The choice of register map can make a noticeable difference to the performance of generated code. We discuss the selection of optimal register maps at translation time in the next section.

3.3 Register Map Selection

Choosing a good register map is crucial to the quality of translation, and moreover the best code may require changing the register map from one code point to the next. Thus, the best register map is the one that minimizes the cost of the peephole translation rule (generates the fastest code) plus any cost of switching register maps from the previous program point—because switching register maps requires adding register move instructions to the generated code to realize the switch at run-

PowerPC Sequence	Live Registers	State Map	x86 Instruction Sequence
mr r1,r2	r1,r2	r1→eax r2→M_1	movl M_1,eax
lwz r1,(r2)	r1,r2	r1→eax r2→ecx	mov (ecx),eax bswap eax
lwz r1,(r2) stw r1,(r3)	r1,r2, r3	r1→eax r2→ecx r3→edx	movl (ecx),eax movl eax,(edx)
mflr r1	r1,lr	r1→eax lr→ecx	movl ecx,eax
lis r1,C0 ori r1,r1,C1	r1	r1→eax	mov $C0C1,eax
subfc r1,r2,r1 adde r1,r1,r3	r1,r2 r3	r1→eax r2→ecx r3→edx	subl ecx,eax adcl edx,eax

Table 3: Examples of peephole translation rules from PowerPC to x86. The x86 sequences are written in AT&T syntax assembly with % signs omitted before registers.

time, switching register maps is not free.

We formulate a dynamic programming problem to choose a minimum cost register map at each program point in a given code region. At each code point, we enumerate all register maps that are likely to produce a translation. Because the space of all register maps is huge, it is important to constrain this space by identifying only the relevant register maps. We consider the register maps at all predecessor code points and extend them using the registers used by the current target sequence. For each register used by the current target sequence, all possibilities of register mappings (after discarding register renamings) are enumerated. Also, we attempt to avoid enumerating register maps that will produce an identical translation to a register map that has already been enumerated, at an equal or higher cost. For each enumerated register map M, the peephole translation table is queried for a matching translation rule T and the corresponding translation cost is recorded. We consider length-1 to length-3 instruction sequences while querying the peephole table for each enumerated register map. Note that there may be multiple possible translations at a given code point, just as there may be multiple valid register maps; we simply keep track of all possibilities. The dynamic programming problem formulation then considers the translation produced for each sequence length while determining the optimal translation for a block of code. Assume for simplicity that the code point under consideration has only one predecessor, and the possible register maps at the predecessor are $P_1, \ldots, P_n$. For simplicity, we also assume that we are translating one instruction at a time and

that T is the minimum cost translation for that instruction if register map M is used. The best cost register map M is then the one that minimizes the cost of switching from a predecessor map P_i to M, the cost of the instruction sequence T, and, recursively, the cost of P_i:

$$cost(M) = cost(T) + min_i(cost(P_i) + switch(P_i, M))$$

To translate multiple source-architecture instructions using a single peephole rule, we extend this approach to consider translations for all length-1 to length-3 sequences that end at the current code point. For example, at the end of instruction i_4 in an instruction sequence (i_1, i_2, i_3, i_4), we search for possible translations for each of the sequences (i_4), (i_3, i_4) and (i_2, i_3, i_4) to find the lowest-cost translation. While considering a translation for the sequence (i_2, i_3, i_4), the predecessor register maps considered are the register maps at instruction i_1. Similarly, the predecessor register maps for sequences (i_3, i_4) and (i_4) are maps at instructions i_2 and i_3 respectively. The cost of register map M is then the minimum among the costs computed for each sequence length.

We solve the recurrence in a standard fashion. Beginning at start of a code region (e.g., a function body), the cost of the preceding register map is initially 0. Working forwards through the code region, the cost of each enumerated register map is computed and stored before moving to the next program point and repeating the computation. When the end of the code region is reached, the register map with the lowest cost is chosen and its decisions are backtracked to decide the register maps and translations at all preceding program points. For program points having multiple predecessors, we use a weighted sum of the switching costs from each predecessor. To handle loops, we perform two iterations of this computation. Interesting examples are too lengthy to include here, but a detailed, worked example of register map selection is in [4].

This procedure of enumerating all register maps and then solving a dynamic programming problem is computationally intensive and, if not done properly, can significantly increase translation time. While the cost of finding the best register map for every code point is not a problem for a static translator, it would add significant overhead to a dynamic translator. To bound the computation time, we prune the set of enumerated register maps at each program point. We retain only the n lowest-cost register maps before moving to the next program point. We allow the value of n to be tunable and refer to it as the *prune size*. We also have the flexibility to trade computation time for lower quality solutions. For example, for code that is not performance critical we can consider code regions of size one (e.g., a single instruction) or even use a fixed register map. In Section 6 we show that the cost of computing the best register maps for frequently executed instructions is very small for our benchmarks. We also discuss the performance sensitivity of our benchmarks to the prune size.

4 Other Issues

In this section, we discuss the main issues relevant to our approach to binary translation.

4.1 Static vs Dynamic Translation

Binary translation can either be performed statically (compile-time) or dynamically (runtime). Most existing tools perform binary translation dynamically for its primary advantage of having a complete view of the current machine state. Moreover, dynamic binary translation provides additional opportunities for runtime optimizations. The drawback of dynamic translation is the overhead of performing translation and book-keeping at runtime. A static translator translates programs offline and can apply more extensive (and potentially whole program) optimizations. However, performing faithful static translation is a slightly harder problem since no assumptions can be made about the runtime state of the process.

Our binary translator is static, though we have avoided including anything in our implementation that would make it impractical to develop a dynamic translator (e.g., whole-program analysis or optimizations) using the same algorithms. Most of the techniques we discuss are equally applicable in both settings and, when they are not, we discuss the two separately.

4.2 Endianness

If the source and destination architectures have different endianness, we convert all memory reads to destination endianness and all memory writes to source endianness. This policy ensures that memory is always in source endianness while registers have destination endianness. The extra byte-swap instructions needed to maintain this invariant are only needed on memory accesses; in particular, we avoid the additional overhead of shuffling bytes on register operations.

While dealing with source-destination architecture pairs with different endianness, special care is required in handling OS-related data structures. In particular, all executable headers, environment variables and program arguments in the program's address space need to be converted from destination endianness to source endianness before transferring control to the translated program. This step is necessary because the source program assumes source endianness for everything while the OS

writes the data structures believing that the program assumes destination endianness. In a dynamic translator, these conversions are performed inside the translator at startup. In a static translator, special initialization code is emitted to perform these conversions at runtime.

4.3 Control Flow Instructions

Like all other opcodes, control flow instructions are also translated using peephole rules. Direct jumps in the source are translated to direct jumps in the translated code, with the jump destination being appropriately adjusted to point to the corresponding translated code. Our superoptimizer is capable of automatically learning translations involving direct jump instructions.

To handle conditional jumps, the condition codes of the source architecture need to be faithfully represented in the destination architecture. Handling condition codes correctly is one of the more involved aspects of binary translation because of the divergent condition-code representations used by different architectures. We discuss our approach to handling condition codes in the context of our PowerPC-x86 binary translator; see Section 5.3. The handling of indirect jumps is more involved and is done differently for static and dynamic translators. We discuss this in detail in Section 5.4.

4.4 System Calls

When translating across two different operating systems, each source OS system call needs to be emulated on the destination OS. Even when translating across the same operating system on different architectures, many system calls require special handling. For example, some system calls are only implemented for specific architectures. Also, if the two architectures have different endianness, proper endianness conversions are required for all memory locations that the system call could read or write. There are other relevant issues to binary translation that we do not discuss here: full system vs. user-level emulation, address translation, precise exceptions, misaligned memory accesses, interprocess communication, signal handling, etc. These problems are orthogonal to the issues in peephole binary translation and our solutions to these issues are standard. In this paper, our focus is primarily on efficient code-generation.

5 Implementation

We have implemented a binary translator that allows PowerPC/Linux executables to run in an x86/Linux environment. The translator is capable of handling almost all PowerPC opcodes (around 180 in all). We have tested our implementation on a variety of different executables and libraries.

The translator has been implemented in C/C++ and O'Caml [13]. Our superoptimizer is capable of automatically inferring peephole translation rules from PowerPC to x86. Because we cannot execute both the target sequence and the enumerated sequence on the same machine, we use a PowerPC emulator (we use Qemu in our experiments) to execute the target sequence. Recall from Section 3.1 that there are two steps to determining which, if any, target instruction sequences are equivalent to the enumerated instruction sequence: first a fast execution test is used to eliminate all but few plausible candidates, and then a complete equivalence check is done by converting both instruction sequences to boolean formulas and deciding a satisfiability query. We use zChaff [15, 26] as our backend SAT solver. We have translated most, but not all, Linux PowerPC system calls. We present our results using a static translator that produces an x86 ELF 32-bit binary executable from a PowerPC ELF 32-bit binary. Because we used the static peephole superoptimizer described in [5] as our starting point, our binary translator is also static, though as discussed previously our techniques could also be applied in a dynamic translator. A consequence of our current implementation is that we also translate all the shared libraries used by the PowerPC program.

In this section, we discuss issues specific to a PowerPC-x86 binary translator. While there exist many architecture-specific issues (as we discuss in this section), the vast bulk of the translation and optimization complexity is still hidden by the superoptimizer.

5.1 Endianness

PowerPC is a big-endian architecture while x86 is a little-endian architecture, which we handle using the scheme outlined in Section 4.2. For integer operations, there exist three operand sizes in PowerPC: 1, 2 and 4 bytes. Depending on the operand size, the appropriate conversion code is required when reading from or writing to memory. We employ the convenient `bswap` x86 instruction to generate efficient conversion code.

5.2 Stack and Heap

On Linux, the stack is initialized with `envp`, `argc` and `argv` and the stack pointer is saved to a canonical register at load time. On x86, the canonical register storing the stack pointer is `esp`; on PowerPC, it is `r1`. When the translated executable is loaded in an x86 environment (in the case of dynamic translation, when the translator is loaded), the `esp` register is initialized to the stack pointer by the operating system while the emulated `r1` register is

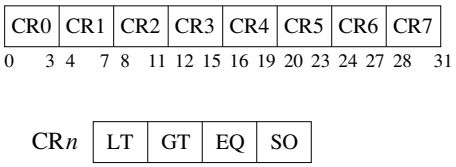


Figure 1: PowerPC architecture has support for eight independent sets of condition codes `CR0-CR7`. Each 4-bit `CR`*n* register uses one bit each to represent less than (`LT`), greater (`GT`), equal (`EQ`) and overflow-summary (`SO`). Explicit instructions are required to read/write the condition code bits.

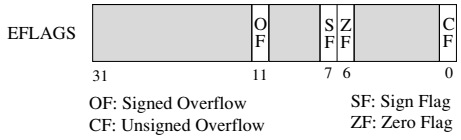


Figure 2: The x86 architecture supports only a single set of condition codes represented as bits in a 32-bit `EFLAGS` register. Almost all x86 instructions overwrite these condition codes.

left uninitialized. To make the stack visible to the translated PowerPC code, we copy the `esp` register to the emulated `r1` register at startup. In dynamic translation, this is done by the translator; in static translation, this is done by the initialization code. The handling of the heap requires no special effort since the `brk` Linux system call used to allocate heap space is identical on both x86 and PowerPC.

5.3 Condition Codes

Condition codes are bits representing quantities such as carry, overflow, parity, less, greater, equal, etc. PowerPC and x86 handle condition codes very differently. Figures 1 and 2 show how condition codes are represented in PowerPC and x86 respectively.

While PowerPC condition codes are written using separate instructions, x86 condition codes are overwritten by almost all x86 instructions. Moreover, while PowerPC compare instructions explicitly state whether they are doing a signed or an unsigned comparison and store only one result in their flags, x86 compare instructions perform both signed and unsigned comparisons and store both results in their condition bits. On x86, the branch instruction then specifies which comparison it is interested in (signed or unsigned). We handle these differences by allowing the PowerPC condition registers (`cr0-cr7`) to be mapped to x86 flags in the register map. For example, an entry `cr0`→`SF` in the register map specifies that, at that program point, the contents of register `cr0` are encoded in the x86 signed flags (`SF`). The translation of a branch instruction then depends on whether the condition register being used ($\texttt{cr}_i$) is mapped to signed (`SF`) or unsigned (`UF`) flags.

5.4 Indirect Jumps

Jumping to an address in a register (or a memory location) is an *indirect* jump. Function pointers, dynamic loading, and case statements are all handled using indirect jumps. Since an indirect jump could jump almost anywhere in the executable, it requires careful handling. Moreover, because the destination of the indirect jump could assume a different register-map than the current one, the appropriate conversion needs to be performed before jumping to the destination. Different approaches for dealing with indirect jumps are needed in static and dynamic binary translators.

Handling an indirect jump in a dynamic translator is simpler. Here, on encountering an indirect jump, we relinquish control to the translator. The translator then performs the register map conversion before transferring control to the (translated) destination address.

Handling an indirect jump in a static translator is more involved. We first identify all instructions that can be possible indirect jump targets. Since almost all well-formed executables use indirect jumps in only a few different code paradigms, it is possible to identify possible indirect jump targets by scanning the executable. We scan the read-only data sections, global offset tables and instruction immediate operands and use a set of pattern matching rules to identify possible indirect jump targets. A lookup table is then constructed to map these jump targets (which are source architecture addresses) to their corresponding destination architecture addresses. However, as we need to perform register map conversion before jumping to the destination address at runtime, we replace the destination addresses in the lookup table with the address of a code fragment that performs the register-map conversion before jumping to the destination address.

The translation of an indirect jump involves a table lookup and some register-map conversion code. While the table lookup is fast, the register-map conversion may involve multiple memory accesses. Hence, an indirect jump is usually an expensive operation.

Although the pattern matching rules we use to identify possible indirect jump targets have worked extremely well in practice, they are heuristics and are prone to adversarial attacks. It would not be difficult to construct an executable that exploits these rules to cause a valid PowerPC program to crash on x86. Hence, in an adver-

ppc	x86	Comparison
`bl`	`call`	`bl` (branch-and-link) saves the instruction pointer to register `lr` while `call` pushes it to stack
`blr`	`ret`	`blr` (branch-to-link-register) jumps to the address pointed-to by `lr`, while `ret` pops the instruction pointer from the stack and jumps to it

Table 4: Function call and return instructions in PowerPC and x86 architectures

sarial scenario, it would be wise to assume that all code addresses are possible indirect jump targets. Doing so results in a larger lookup table and more conversion code fragments, increasing the overall size of the executable, but will have no effect on running time apart from possible cache effects.

5.5 Function Calls and Returns

Function calls and returns are handled in very different ways in PowerPC and x86. Table 4 lists the instructions and registers used in function calls and returns for both architectures.

We implement function calls of the PowerPC architecture by simply emulating the link-register (`lr`) like any other PowerPC register. On a function call (`bl`), the link register is updated with the value of the next PowerPC instruction pointer. A function return (`blr`) is treated just like an indirect jump to the link register.

The biggest advantage of using this scheme is its simplicity. However, it is possible to improve the translation of the `blr` instruction by exploiting the fact that `blr` is always used to return from a function. For this reason, it is straightforward to predict the possible jump targets of `blr` at translation time (it will be the instruction following the function call `bl`). At runtime, the value of the link register can then be compared to the predicted value to see if it matches, and then jump accordingly. This information can be used to avoid the extra memory reads and writes required for register map conversion in an indirect jump. We have implemented this optimization; while this optimization provides significant improvements while translating small recursive benchmarks (e.g., recursive computation of the fibonacci series), it is not very effective for larger benchmarks (e.g., SPEC CINT2000).

5.6 Register Name Constraints

Another interesting challenge while translating from PowerPC to x86 is dealing with instructions that operate only on specific registers. Such instructions are present on both PowerPC and x86. Table 5 shows some such x86 instructions.

Opcode	Registers	Description
`mul` *reg32*	`eax, edx`	Multiplies *reg32* with eax and stores the 64-bit result in `edx:eax`.
`div` *reg32*	`eax, edx`	Divides `edx:eax` by *reg32* and stores result in `eax`.
any 8-bit insn	`eax, ebx` `ecx, edx`	8-bit operations can only be performed on these four registers.

Table 5: Examples of x86 instructions that operate only on certain fixed registers.

To be able to generate peephole translations involving these special instructions, the superoptimizer is made aware of the constraints on their operands during enumeration. If a translation is found by the superoptimizer involving these special instructions, the generated peephole rule encodes the name constraints on the operands as *register name constraints*. These constraints are then used by the translator at code generation time.

5.7 Self-Referential and Self-Modifying Code

We handle self-referential code by leaving a copy of the source architecture code in its original address range for the translated version. To deal with self-modifying code and dynamic loading, we would need to invalidate the translation of a code fragment on observing any modification to that code region. To do this, we would trap any writes to code regions and perform the corresponding invalidation and re-translation. For a static translator, this involves making the translator available as a shared library—a first step towards a full dynamic translator. While none of our current benchmarks contain self-modifying code, it would be straightforward to extend our translator to handle such scenarios.

5.8 Untranslated Opcodes

For 16 PowerPC opcodes our translator failed to find a short equivalent x86 sequence of instructions automatically. In such cases, we allow manual additions to the peephole table. Table 6 describes the number and types of hand additions: 9 are due to instructions involving indirect jumps and 7 are due to complex PowerPC instructions that cannot be emulated using a bounded length straight-line sequence of x86 instructions. For some

Number of Additions	Reason
2	Overflow/underflow semantics of the divide instruction (`div`)
2	Overflow semantics of `srawi` shift instruction
1	The rotate instruction `rlwinm`
1	The `cntlzw` instruction
1	The `mfcr` instruction
9	Indirect jumps referencing the jumptable

Table 6: The distribution of the manual translation rules we added to the peephole translation table.

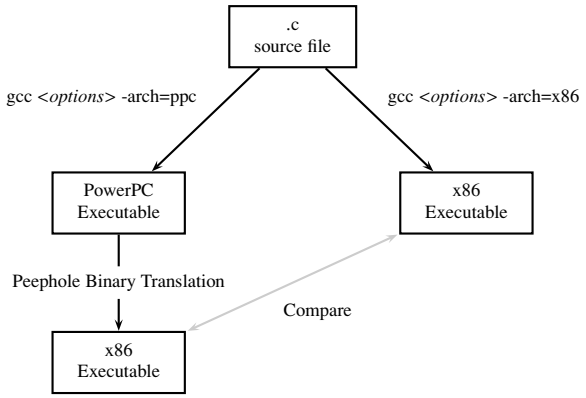


Figure 3: Experimental Setup. The translated binary executable is compared with the natively-compiled x86 executable. While comparing, the same compiler optimization options are used on both branches.

more complex instructions mostly involving interrupts and other system-related tasks, we used the slow but simple approach of emulation using C-code.

5.9 Compiler Optimizations

An interesting observation while doing our experiments was that certain compiler optimizations often have an adverse effect on the performance of our binary translator. For example, an optimized PowerPC executable attempts to use all 8 condition-registers (`cr0-cr7`). However, since x86 has only one set of flags, other condition registers need to be emulated using x86 registers causing extra register pressure. Another example of an unfriendly compiler optimization is instruction scheduling. An optimizing PowerPC compiler separates two instructions involving a data dependency to minimize pipeline stalls, while our binary translator would like the data-dependent instructions to be together to allow the superoptimizer to suggest more aggressive optimizations. Our implementation reorders instructions within basic blocks to minimize the length of dependencies prior to translation.

6 Experimental Results

We performed our experiments using a Linux machine with a single Intel Pentium 4 3.0GHz processor, 1MB cache and 4GB of memory. We used `gcc` version 4.0.1 and `glibc` version 2.3.6 to compile the executables on both Intel and PowerPC platforms. To produce identical compilers, we built the compilers from their source tree using exactly the same configuration options for both architectures. While compiling our benchmarks, we used the `-msoft-float` flag in `gcc` to emulate floating point operations in software; our translator currently does not translate floating point instructions. For all our benchmarks except one, emulating floating point in software makes no difference in performance. All the executables were linked statically and hence, the libraries were also converted from PowerPC to x86 at translation time. To emulate some system-level PowerPC instructions, we used C-code from the open source emulator Qemu [17].

In our experiments, we compare the executable produced by our translator to a natively-compiled executable. The experimental setup is shown in Figure 3. We compile from the C source for both PowerPC and x86 platforms using `gcc`. The same compiler optimization options are used for both platforms. The PowerPC executable is then translated using our binary translator to an x86 executable. And finally, the translated x86 executable is compared with the natively-compiled one for performance.

One would expect the performance of the translated executable to be strictly lower than that of the natively-compiled executable. To get an idea of the state-of-the-art in binary translation, we discuss two existing binary translators. A general-purpose open-source emulator, Qemu [17], provides 10–20% of the performance of a natively-compiled executable (i.e., 5–10x slowdown). A recent commercially available tool by Transitive Corporation [22] (which is also the basis of Apple's Rosetta translator) claims "typically about 70–80%" of the performance of a natively-compiled executable on their website [18]. Both Qemu and Transitive are dynamic binary translators, and hence Qemu and Rosetta results include the translation overhead, while the results for our static translator do not. We estimate the translation overhead of our translator in Section 6.1.

Table 7 shows the performance of our binary translator on small compute-intensive microbenchmarks. (All

Benchmark	Description	-O0	-O2	-O2ofp
emptyloop	A bounded for-loop doing nothing	98.56 %	128.72 %	127 %
fibo	Compute first few Fibonacci numbers	118.90 %	319.13 %	127.78 %
quicksort	Quicksort on 64-bit integers	81.36 %	92.61 %	90.23 %
mergesort	Mergesort on 64-bit integers	83.22 %	91.54 %	84.35 %
bubblesort	Bubble-sort on 64-bit integers	75.12 %	70.92 %	64.86 %
hanoi1	Towers of Hanoi Algorithm 1	84.83 %	70.03 %	61.96 %
hanoi2	Towers of Hanoi Algorithm 2	107.14 %	139.64 %	143.69 %
hanoi3	Towers of Hanoi Algorithm 3	81.04 %	90.14 %	80.15 %
traverse	Traverse a linked list	69.06 %	67.67 %	67.15 %
binsearch	Perform binary search on a sorted array	65.38 %	61.24 %	62.15 %

Table 7: Performance of the binary translator on some compute-intensive microbenchmarks. The columns represent the optimization options given to gcc. '-O2ofp' expands to '-O2 -fomit-frame-pointer'. '-O2ofp' omits storing the frame pointer on x86. On PowerPC, '-O2ofp' is identical to '-O2'. The performance is shown relative to a natively compiled application (the performance of a natively compiled application is 100%).

	O0			O2		
	native (secs)	peep (secs)	% of native	native (secs)	peep (secs)	% of native
bzip2	311	470	66.2 %	195	265	73.7 %
gap	165	313	52.5 %	87	205	42.5 %
gzip	264	398	66.3 %	178	315	56.5 %
mcf	193	221	87.3 %	175	184	94.7 %
parser	305	520	58.7 %	228	338	67.3 %
twolf	2184	1306	167.2 %	1783	1165	153.0 %
vortex	193	463	41.7 %	161	-	-

Table 8: Performance of the binary translator on SPEC CINT2000 benchmark applications. The percentage (% of native) columns represent performance relative to the x86 performance (the performance of a natively compiled application is 100%). '-' entries represent failed translations.

reported runtimes are computed after running the executables at least 3 times.) Our microbenchmarks use three well-known sorting algorithms, three different algorithms to solve the Towers of Hanoi, one benchmark that computes the Fibonacci sequence, a link-list traversal, a binary search on a sorted array, and an empty for-loop. All these programs are written in C. They are all highly compute-intensive and hence designed to stress-test the performance of binary translation.

The translated executables perform roughly at 90% of the performance of a natively-compiled executable on average. Some benchmarks perform as low as 64% of native performance and many benchmarks outperform the natively compiled executable. The latter result is a bit surprising. For unoptimized executables, the binary translator often outperforms the natively compiled executable because the superoptimizer performs optimizations that are not seen in an unoptimized natively compiled executable. The bigger surprise occurs when the translated executable outperforms an already optimized executable (columns -O2 and -O2ofp) indicating that even mature optimizing compilers today are not producing the best possible code. Our translator sometimes outperforms the native compiler for two reasons:

- The gcc-generated code for PowerPC is sometimes superior to the code generated for x86. This situation is in line with the conventional wisdom that it is easier to write a RISC optimizer than a CISC optimizer.
- Because we search the space of all possible translations while performing register mapping and instruction-selection, the code generated by our translator is often superior to that generated by gcc.

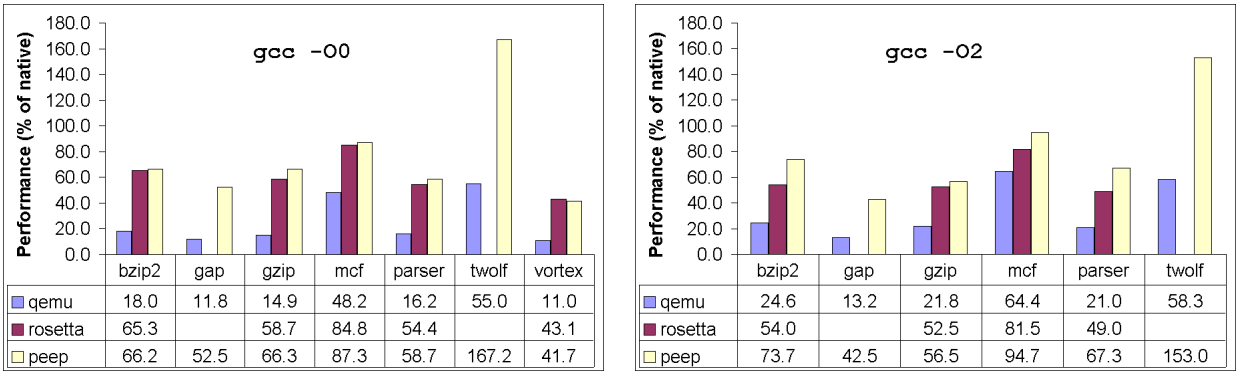


Figure 4: Performance comparison of our translator (`peep`) with open source binary translator Qemu (`qemu`), and a commercial binary translator Apple Rosetta (`rosetta`). The bars represent performance relative to a natively compiled executable (higher is better). Missing bars are due to failed translations.

When compared with Apple Rosetta, our translator consistently performs better than Rosetta on all these microbenchmarks. On average, our translator is 170% faster than Apple Rosetta on these small programs.

A striking result is the performance of the `fibo` benchmark in the `-O2` column where the translated executable is almost three times faster than the natively-compiled and optimized executable. On closer inspection, we found that this is because `gcc`, on x86, uses one dedicated register to store the frame pointer by default. Since the binary translator makes no such reservation for the frame pointer, it effectively has one extra register. In the case of `fibo`, the extra register avoids a memory spill present in the natively compiled code causing the huge performance difference. Hence, for a more equal comparison, we also compare with the '`-fomit-frame-pointer`' `gcc` option on x86 (`-O2ofp` column).

Table 8 gives the results for seven of the SPEC integer benchmarks. (The other benchmarks failed to run correctly due to the lack of complete support for all Linux system calls in our translator). Figure 4 compares the performance of our translator to Qemu and Rosetta. In our comparisons with Qemu, we used the same PowerPC and x86 executables as used for our own translator. For comparisons with Rosetta, we could not use the same executables, as Rosetta supports only Mac executables while our translator supports only Linux executables. Therefore, to compare, we recompiled the benchmarks on Mac to measure Rosetta performance. We used exactly the same compiler version (`gcc` 4.0.1) on the two platforms (Mac and Linux). We ran our Rosetta experiments on a Mac Mini Intel Core 2 Duo 1.83GHz processor, 32KB L1-Icache, 32KB L1-Dcache, 2MB L2-cache and 2GB of memory. These benchmarks spend very little time in the kernel, and hence we do not expect any bias in results due to differences in the two operating systems. The differences in the hardware could cause some bias in the performance comparisons of the two translators. While it is hard to predict the direction and magnitude of this bias, we expect it to be insignificant.

Our peephole translator fails on `vortex` when it is compiled using `-O2`. Similarly, Rosetta fails on `twolf` for both optimization options. These failures are most likely due to bugs in the translators. We could not obtain performance numbers for Rosetta on `gap` because we could not successfully build `gap` on Mac OS X. Our peephole translator achieves a performance of 42–164% of the natively compiled executable. Comparing with Qemu, our translator achieves 1.3–4x improvement in performance. When compared with Apple Rosetta, our translator performs 12% better (average) on the executables compiled with `-O2` and 3% better on the executables compiled with `-O0`. Our system performs as well or better than Rosetta on almost all our benchmarks, the only exceptions being `-O0` for `vortex` where the peephole translator produces code 1.4% slower than Rosetta, and `-O2` for `vortex`, which the peephole translator fails to translate. The median performance of the translator on these compute-intensive benchmarks is 67% of native code.

A very surprising result is the performance of the `twolf` benchmark where the performance of our translator is significantly better than the performance of natively compiled code. On further investigation, we found that `twolf`, when compiled with `-msoft-float`, spends a significant fraction of time ($\sim 50\%$) in the floating point emulation library (which is a part of `glibc`). The x86 floating point emulation library functions contain a redundant function call to determine the current instruction pointer, while the PowerPC floating point emulation code contains no such function call. This is the

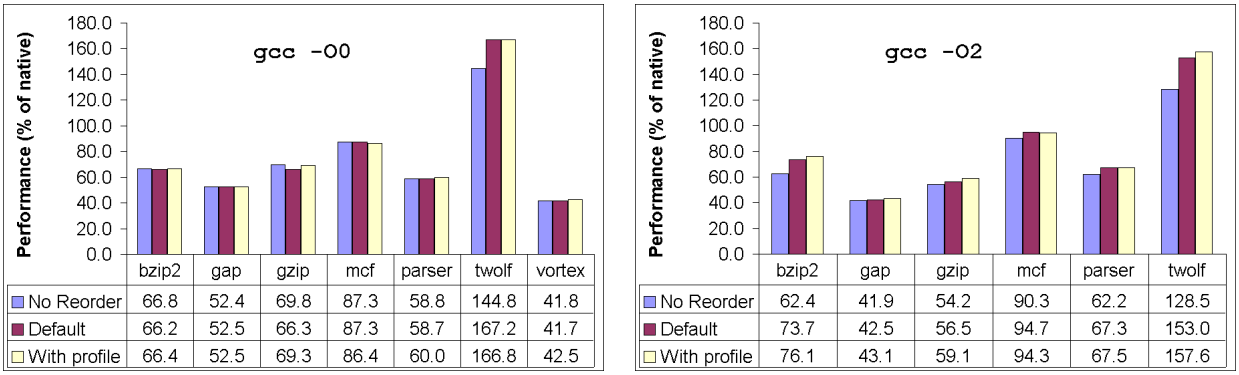


Figure 5: Performance comparison of the default peephole translator with variants `No-Reorder` and `With-Profile`. The bars represent performance relative to a natively compiled executable (higher is better).

default `glibc` behavior and we have not found a way to change it. Coupled with the optimizations produced by our translator, this extra overhead in natively compiled x86 code leads to better overall performance for translated code. We do not see this effect in all our other benchmarks as they spend an insignificant fraction ($< 0.01\%$) of time in floating point emulation. The complete data on the running times of natively compiled and translated benchmarks is available in [4].

Next, we consider the performance of our translator on SPEC benchmarks by toggling some of the optimizations. The purpose of these experiments is to obtain insight into the performance impact of these optimizations. We consider two variants of our translator:

1. `No-Reorder`: Recall that, by default, we cluster data-dependent instructions inside a basic block for better translation (refer Section 5.9). In this variant, we turn off the re-ordering of instructions.

2. `With-Profile`: In this variant, we profile our executables in a separate offline run and record the profiling data. Then, we use this data to determine appropriate weights of predecessors and successors during register map selection (see Section 3.3).

Figure 5 shows the comparisons of the two variants relative to the default configuration. We make two key observations:

- The re-ordering of instructions inside a basic block has a significant performance impact on executables compiled with `-O2`. The PowerPC optimizing compiler separates data-dependent instructions to minimize data stalls. To produce efficient translated code, it helps to "de-optimize" the code by bringing data-dependent instructions back together. On average, the performance gain by re-ordering instructions inside a basic block is 6.9% for `-O2` executables. For `-O0` executables, the performance impact of re-ordering instructions is negligible, except `twolf` where a significant fraction of time is spent in precompiled optimized libraries.

- From our results, we think that profiling information can result in small but notable improvements in performance. In our experiments, the average improvement obtained by using profiling information is 1.4% for `-O2` executables and 0.56% for `-O0` executables. We believe our translator can exploit such runtime profiling information in a dynamic binary translation scenario.

Our superoptimizer uses a peephole size of at most 2 PowerPC instructions. The x86 instruction sequence in a peephole rule can be larger and is typically 1–3 instructions long. Each peephole rule is associated with a cost that captures the approximate cycle cost of the x86 instruction sequence.

We compute the peephole table offline only once for every source-destination architecture pair. The computation of the peephole table can take up to a week on a single processor. On the other hand, applying the peephole table to translate an executable is fast (see Section 6.1). For these experiments, the peephole table consisted of approximately 750 translation rules. Given more time and resources, it is straightforward to scale the number of peephole rules by running the superoptimizer on longer length sequences. More peephole rules are likely to give better performance results.

The size of the translated executable is roughly 5–6x larger than the source PowerPC executable. Of the total size of the translated executable, roughly 40% is occupied by the translated code, 20% by the code and data sections of the original executable, 25% by the indirect jump lookup table and the remaining 15% by other management code and data. For our benchmarks, the average size of the code sections in the original PowerPC executables is around 650 kilobytes, while the average

size of the code sections in the translated executables is around 1400 kilobytes. Because both the original and translated executables operate on the same data and these benchmarks spend more than 99.99% of their time in less than 2% of the code, we expect their working set sizes to be roughly the same.

6.1 Translation Time

Translation time is a significant component of the runtime overhead for dynamic binary translators. As our prototype translator is static, we do not account for this overhead in the experiments in Section 6. In this section we analyze the time consumed by our translator and how it would fit in a dynamic setting.

Our static translator takes 2–6 minutes to translate an executable with around 100K instructions, which includes the time to disassemble a PowerPC executable, compute register liveness information for each function, perform the actual translation including computing the register map for each program point (see Section 3.3), build the indirect jump table and then write the translated executable back to disk. Of these various phases, computing the translation and register maps accounts for the vast majority of time.

A dynamic translator, on the other hand, typically translates instructions when, and only when, they are executed. Thus, no time is spent translating instructions that are never executed. Because most applications use only a small portion of their extensive underlying libraries, in practice dynamic translators only translate a small part of the program. Moreover, dynamic translators often trade translation time for code quality, spending more translation time and generating better code for hot code regions.

To understand the execution characteristics of a typical executable, we study our translator's performance on `bzip2` in detail. (Because all of our applications build on the same standard libraries, which form the overwhelming majority of the code, the behavior of the other applications is similar to `bzip2`.) Of the 100K instructions in `bzip2`, only around 8–10K instructions are ever executed in the benchmark runs. Of these, only around 2K instructions (hot regions) account for more than 99.99% of the execution time. Figure 6 shows the time spent in translating the hot regions of code using our translator.

We plot the translation time with varying prune sizes; because computing the translation and register maps dominate, the most effective way for our system to trade code quality for translation speed is by adjusting the prune size (recall Section 3.3). We also plot the performance of the translated executable at these prune sizes. At prune size 0, an arbitrary register map is chosen where

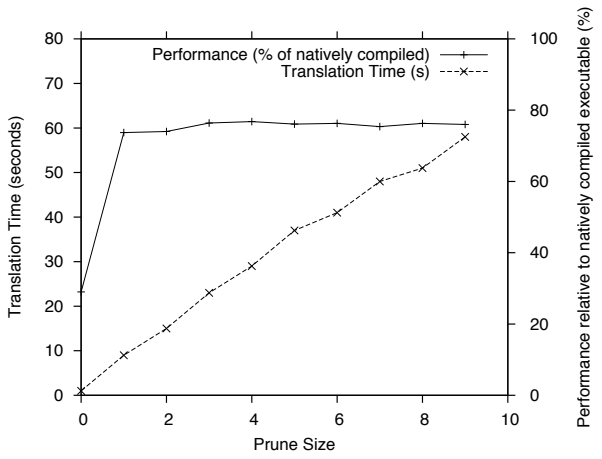


Figure 6: Translation time overhead with varying prune size for `bzip2`.

all PowerPC registers are mapped to memory. At this point, the translation time of the hot regions is very small (less than 0.1 seconds) at the cost of the execution time of the translated executable. At prune size 1 however, the translation time increases to 8 seconds and the performance already reaches 74% of native. At higher prune sizes, the translation overhead increases significantly with only a small improvement in runtime (for `bzip2`, the runtime improvement is 2%). This indicates that even at a small prune size (and hence a low translation time), we obtain good performance. While higher prune sizes do not significantly improve the performance of the translator on SPEC benchmarks, they make a significant difference to the performance of tight inner loops in some of our microbenchmarks.

Finally, we point out that while the translation cost reported in Figure 6 accounts for only the translation of hot code regions, we can use a fast and naive translation for the cold regions. In particular, we can use an arbitrary register map (prune size of 0) for the rarely executed instructions to produce fast translations of the remaining code; for `bzip2` it takes less than 1 second to translate the cold regions using this approach. Thus we estimate that a dynamic translator based on our techniques would require under 10 seconds in total to translate `bzip2`, or less than 4% of the 265 seconds of run-time reported in Table 8.

7 Related Work

Binary translation first became popular in the late 1980s as a technique to improve the performance of existing emulation tools. Some of the early commercial binary translators were those by Hewlett-Packard to migrate their customers from its HP 3000 line to the new Precision architecture (1987), by Digital Equipment Corpo-

ration to migrate users of VAX, MIPS, SPARC and x86 to Alpha (1992), and by Apple to run Motorola 68000 programs on their PowerMAC machines(1994).

By the mid-1990's more binary translators had appeared: IBM's DAISY [8] used hardware support to translate popular architectures to VLIW architectures, Digital's FX!32 ran x86/WinNT applications on Alpha/WinNT [7], Ardi's Executor [9] ran old Macintosh applications on PCs, Sun's Wabi [21] executed Microsoft Windows applications in UNIX environments and Embra [24], a machine simulator, simulated the processors, caches and other memory systems of uniprocessors and cache-coherent multiprocessors using binary translation. A common feature in all these tools is that they were all designed to solve a specific problem and were tightly coupled to the source and/or destination architectures and operating systems. For this reason, no meaningful performance comparisons exist among these tools.

More recently, the moral equivalent of binary translation is used extensively in Java just-in-time (JIT) compilers to translate Java bytecode to the host machine instructions. This approach is seen as an efficient solution to deal with the problem of portability. In fact, some recent architectures especially cater to Java applications as these applications are likely to be their first adopters [2].

An early attempt to build a general purpose binary translator was the UQBT framework [23] that described the design of a machine-adaptable dynamic binary translator. The design of the UQBT framework is shown in Figure 7. The translator works by first decoding the machine-specific binary instructions to a higher level RTL-like language (RTL stands for register transfer lists). The RTLs are optimized using a machine-independent optimizer, and finally machine code is generated for the destination architecture from the RTLs. Using this approach, UQBT had up to a 6x slowdown in their first implementation. A similar approach has been taken by a commercial tool being developed at Transitive Corporation [22]. Transitive first disassembles and decodes the source instructions to an intermediate language, performs optimizations on the intermediate code and finally assembles it back to the destination architecture. In their current offerings, Transitive supports SPARC-x86, PowerPC-x86, SPARC-x86/64-bit and SPARC-Itanium source-destination architecture pairs.

A potential weakness in the approach used by UQBT and Transitive is the reliance on a well-designed intermediate RTL language. A universal RTL language would need to capture the peculiarities of all different machine architectures. Moreover, the optimizer would need to understand these different language features and be able to exploit them. It is a daunting task to first design a good and universal intermediate language and then write an optimizer for it, and we believe using a single intermediate language is hard to scale beyond a few architectures. Our comparisons with Apple Rosetta (Transitive's PowerPC-x86 binary translator) suggest that superoptimization is a viable alternative and likely to be easier to scale to many machine pairs.

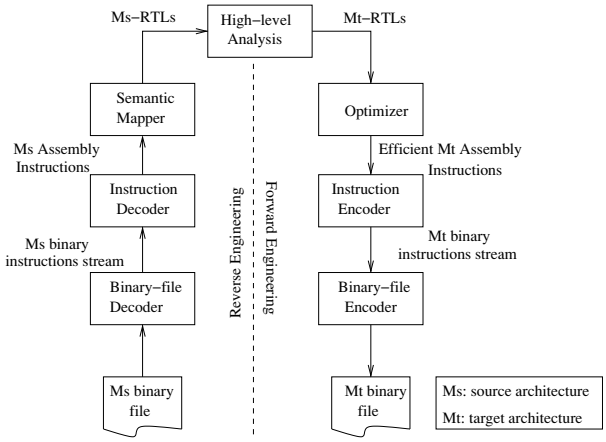


Figure 7: The framework used in UQBT binary translation. A similar approach is taken by Transitive Corporation.

In recent years, binary translation has been used in various other settings. Intel's IA-32 EL framework provides a software layer to allow running 32-bit x86 applications on IA-64 machines without any hardware support. Qemu [17] uses binary translation to emulate multiple source-destination architecture pairs. Qemu avoids dealing with the complexity of different instruction sets by encoding each instruction as a series of operations in C. This allows Qemu to support many source-destination pairs at the cost of performance (typically 5–10x slowdown). Transmeta Crusoe [12] uses on-chip hardware to translate x86 CISC instructions to RISC operations on-the-fly. This allows them to achieve comparable performance to Intel chips at lower power consumption. Dynamo and Dynamo-RIO [3, 6] use dynamic binary translation and optimization to provide security guarantees, perform runtime optimizations and extract program trace information. Strata [19] provides a software dynamic translation infrastructure to implement runtime monitoring and safety checking.

8 Conclusions and Future Work

We present a scheme to perform efficient binary translation using a superoptimizer that automatically learns translations from one architecture to another. We demonstrate through experiments that our superoptimization-based approach results in competitive performance while

significantly reducing the complexity of building a high performance translator by hand.

We have found that this approach of first learning several peephole translations in an offline phase and then applying them to simultaneously perform register mapping and instruction selection produces an efficient code generator. In future, we wish to apply this technique to other applications of code generation, such as just-in-time compilation and machine virtualization.

9 Acknowledgments

We thank Suhabe Bugrara, Michael Dalton, Adam Oliner and Pramod Sharma for reviewing and giving valuable feedback on earlier drafts of the paper. We also thank Ian Pratt (our shepherd) and the anonymous reviewers.

References

[1] Apple Rosetta. `http://www.apple.com/rosetta/`.

[2] Azul Systems. `http://www.azulsystems.com/`.

[3] BALA, V., DUESTERWALD, E., AND BANERJIA, S. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices 35*, 5 (2000), 1–12.

[4] BANSAL, S. *Peephole Superoptimization*. PhD thesis, Stanford University, 2008.

[5] BANSAL, S., AND AIKEN, A. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (October 21-25, 2006), pp. 394–403.

[6] BRUENING, D. *Efficient, Transparent and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[7] CHERNOFF, A., HERDEG, M., HOOKWAY, R., REEVE, C., RUBIN, N., TYE, T., YADAVALLI, S. B., AND YATES, J. FX!32: A profile-directed binary translator. *IEEE Micro 18*, 2 (Mar/Apr 1998), 56–64.

[8] EBCIOGLU, K., AND ALTMAN, E. R. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)* (1997), pp. 26–37.

[9] Executor by ARDI. `http://en.wikipedia.org/wiki/Executor_(software)`.

[10] GRANLUND, T., AND KENNER, R. Eliminating branches using a superoptimizer and the gnu C compiler. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation* (June 1992), vol. 27, pp. 341–352.

[11] HALFHILL, T. Transmeta breaks x86 low-power barrier. *Microprocessor Report* (February 2000).

[12] KLAIBER, A. The technology behind Crusoe processors. Tech. rep., Transmeta Corp., January 2000.

[13] LEROY, X., DOLIGEZ, D., GARRIGUE, J., AND VOUILLON, J. The Objective Caml system. Software and documentation available at `http://caml.inria.fr`.

[14] MASSALIN, H. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)* (1987), pp. 122–126.

[15] MOSKEWICZ, M. W., MADIGAN, C. F., ZHAO, Y., ZHANG, L., AND MALIK, S. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)* (2001), pp. 530–535.

[16] PowerVM Lx86 for x86 Linux applications. `http://www.ibm.com/developerworks/linux/lx86/index.html`.

[17] Qemu. `http://fabrice.bellard.free.fr/qemu/`.

[18] QuickTransit for Power-to-X86. `http://transitive.com/products/pow_x86.htm`.

[19] SCOTT, K., AND DAVIDSON, J. Strata: A software dynamic translation infrastructure. In *IEEE Workshop on Binary Translation* (2001).

[20] Server consolidation and containment with VMware Virtual Infrastructure and Transitive. `http://www.transitive.com/pdf/VMwareTransitiveSolutionBrief.pdf`.

[21] SunSoft Wabi. `http://www.sun.com/sunsoft/Products/PC-Integration-products/`.

[22] Transitive Technologies. `http://www.transitive.com/`.

[23] UNG, D., AND CIFUENTES, C. Machine-adaptable dynamic binary translation. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo)* (2000), pp. 41–51.

[24] WITCHEL, E., AND ROSENBLUM, M. Embra: Fast and flexible machine simulation. In *Proceedings of the ACM SIGMETRICS conference on Measurement and Modeling of Computer Systems* (1996), pp. 68–79.

[25] YANG, B.-S., MOON, S.-M., PARK, S., LEE, J., LEE, S., PARK, J., CHUNG, Y. C., KIM, S., EBCIOGLU, K., AND ALTMAN, E. R. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)* (1999), pp. 128–138.

[26] ZHANG, L., MADIGAN, C. F., MOSKEWICZ, M. W., AND MALIK, S. Efficient conflict driven learning in boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)* (November 2001), pp. 279–285.

R2: An Application-Level Kernel for Record and Replay

Zhenyu Guo[†] Xi Wang[‡] Jian Tang[†] Xuezheng Liu[†]
Zhilei Xu[‡] Ming Wu[†] M. Frans Kaashoek[§] Zheng Zhang[†]
[†]*Microsoft Research Asia* [‡]*Tsinghua University* [§]*MIT CSAIL*

ABSTRACT

Library-based record and replay tools aim to reproduce an application's execution by recording the results of selected functions in a log and during replay returning the results from the log rather than executing the functions. These tools must ensure that a replay run is identical to the record run. The challenge in doing so is that only invocations of a function by the application should be recorded, recording the side effects of a function call can be difficult, and not executing function calls during replay, multithreading, and the presence of the tool may change the application's behavior from recording to replay. These problems have limited the use of such tools.

R2 allows developers to choose functions that can be recorded and replayed correctly. Developers annotate the chosen functions with simple keywords so that R2 can handle calls with side effects and multithreading. R2 generates code for record and replay from templates, allowing developers to avoid implementing stubs for hundreds of functions manually. To track whether an invocation is on behalf of the application or the implementation of a selected function, R2 maintains a mode bit, which stubs save and restore.

We have implemented R2 on Windows and annotated large parts (1,300 functions) of the Win32 API, and two higher-level interfaces (MPI and SQLite). R2 can replay multithreaded web and database servers that previous library-based tools cannot replay. By allowing developers to choose high-level interfaces, R2 can also keep recording overhead small; experiments show that its recording overhead for Apache is approximately 10%, that recording and replaying at the SQLite interface can reduce the log size up to 99% (compared to doing so at the Win32 API), and that using optimization annotations for BitTorrent and MPI applications achieves log size reduction ranging from 13.7% to 99.4%.

1 INTRODUCTION

Replay is a powerful technique for debugging applications. When an application is running, a record and replay tool records all interactions between the application and its environment (e.g., reading input from a file, receiving a message). Then when a developer wants to track down an error, she can replay the application to a given state based on the recorded interactions, and investigate how the application reached that state. Replay is particularly useful in the context of distributed applications with many processes. If one of the processes has an error, the developer can investigate the error by replaying that single process instead of all processes, observing the external interactions in the same order as during the recording.

R2 is a novel record and replay tool that allows developers to choose at what interface the interactions between the application and its environment are recorded and replayed. R2 resides in the application's address space and intercepts all functions in the chosen interface. R2 uses a library-based approach to simplify deployment, compared to hardware or virtual machine based approaches. During recording, R2 executes the intercepted calls and records their results in a log. During replay, the application runs as usual (i.e., making library and systems calls, modifying memory, etc.), but R2 intercepts calls in the chosen interface, prevents the real implementation of the calls from executing, and instead gives the application the results of the calls that were previously recorded in the log.

R2 allows developers to choose the interposed interface for two reasons: correctness and performance. The developers can choose an interface that is easy to make *replay faithful*. If an interface is replay faithful, the replay run of an application is identical to the recorded run of the application. This property ensures that if a problem appears while the application is running in recording mode, the problem will also appear during replay. (R2 does *not* attempt to make the behavior of an application with or without recording identical. That is, if a problem appears in the application while it is not being recorded, R2 does not guarantee that the problem will happen again when the application is recorded.)

Achieving faithful replay can be challenging because only calls on behalf of the application should be recorded, recording the effects of a call can be difficult, and not executing calls during replay, multithreading, the presence of the tool in the application's address space may cause the application to behave differently during replay. Previous library-based tools (e.g., liblog [10] and Jockey [28]) interpose a fixed low-level interface and omit calls that are difficult to make replay faithful, and thus limit the applications they can replay.

Consider recording and replaying at the system call interface, which is a natural choice because the application interacts with its environment through system calls. It is not easy to record the output of all system calls. For

Annotation	Scope	Description	Section
in	parameter	input (read-only) parameter	§ 3
out	parameter	output (mutable) parameter	§ 3
bsize(*val*)	parameter	modified size of an array buffer (*val* can be any expression)	§ 3
xpointer(*kind*)	parameter	address allocated internally (*kind* can be *null*, *thread*, or *process*)	§ 3
prepare(*key*,*buf*)	function	prepare asynchronous data transfer	§ 3
commit(*key*,*size*)	function	commit asynchronous data transfer	§ 3
callback	parameter	callback function pointer (upcall)	§ 4
sync(*key*)	function	causality among syscalls and upcalls (*key* can be any expression)	§ 4
cache	function	cache for reducing log size	§ 6
reproduce	function	reproduce I/O for reducing log size	§ 6

Table 1: Annotation keywords (for data transfer, execution order, and optimization).

example, to ensure faithful replay the developer must arrange to record the results of `socketcall` correctly but its results vary for different parameters. For such cases R2 makes it easy for a developer to choose an interface consisting of higher-level functions that cause the same interactions with the environment, but are easier to record and replay. For example, the developer may choose `recv`, which calls `socketcall`; `recv`'s effects are easier to record and replay.

The second reason for allowing developers to choose the interface is that they can choose an interface that results in low recording overhead for their applications. Low overhead is important because the developers can then run their applications in recording mode even during deployment, which may help in debugging problems that show up rarely. To reduce overhead, a developer might choose to record and replay the interactions at a high-level interface (e.g., MPI and SQL library interface such as SQLite) because less information must be recorded. In addition, these higher-level interfaces may be easier to be replay faithful.

To lower the implementation effort for intercepting, recording, and replaying a chosen interface, R2 generates stubs for the calls in the chosen interface and arranges that these stubs are called when the application invokes the calls. The stubs perform the recording and the replay of the calls. To ensure that these stubs behave in way that is replay faithful, the developer must annotate the interface with simple annotations (see Table 1) that specify, for example, how data is transferred across the interposed interface for calls that change memory in addition to having a return value. To reduce the effort of annotating R2 reuses existing annotations from SAL [13] for Windows API. Inspired by the kernel/user division in operating systems, R2 uses a mode bit, which stubs save and restore, to track if a call is on behalf of the application and should be recorded.

We have implemented R2 on Windows, and used it to record and replay at three interfaces (Win32, MPI, and SQLite API). It has successfully replayed various system applications (see Section 8), including applications that cannot be replayed with previous library-based tools. R2 has also replayed and helped to debug two distributed systems, and has been used as a building block in other tools [20, 31, 22].

The main contributions of the paper are: first, a record and replay tool that allows developers to decide which interface to record and replay; second, a set of annotations that allows strict separation of the application above the interposed interface and the implementation below the interface, and that reduces the manual work that a developer must do; third, an implementation of a record and replay library for Windows, which is capable of replaying challenging system applications with low recording overhead.

The rest of the paper is organized as follows. Section 2 gives an overview of the design. Section 3 and 4 describe the annotations for data transfers and execution orders, respectively. Section 5 discuss how to record and replay the MPI and SQLite interfaces. Section 6 and 7 describe annotations for optimizations and implementation details, respectively. We evaluate R2 in Section 8, discuss related work in Section 9, and conclude in Section 10.

2 Design Overview

A goal of R2 is to replay applications faithfully. To do so the calls to intercept must be carefully chosen and stubs must handle several challenges. This section starts with an example to illustrate the challenges, and then describes how R2 addresses them.

2.1 An Example and Challenges

Faithful replay is particularly challenging for system applications, which interact with the operating system in complicated ways. Consider Figure 1, a typical network program on Windows: a thread binds a socket to an I/O port (`CreateIoCompletionPort`,

```
1  struct iocb {
2      OVERLAPPED ov;
3      void * buf, * user_data;
4  };
5
6  int main() {
7      HANDLE hPort = ...;
8      for (...)
9        CreateThread(..., WorkerThread, hPort, ...);
10     ...
11     SOCKET s = socket(...);
12     CreateIoCompletionPort(s, hPort, ...);
13     struct iocb * cb = (struct iocb *)malloc(...);
14     cb->buf = malloc(BUFSIZ);
15     cb->user_data = ...;
16     BOOL fSucc = ReadFileEx(s, cb->buf, BUFSIZ,
17                             (OVERLAPPED *)&cb, 0);
18     ...
19 }
20
21 DWORD WINAPI WorkerThread(HANDLE hPort) {
22     for ( ; ; ) {
23        struct iocb * cb;
24        DWORD size;
25        GetQueuedCompletionStatus(hPort, &size, ...,
26                                  (OVERLAPPED *)&cb, ...);
27        void * buf = cb->buf;
28        void * user_data = cb->user_data;
29        ...
30     }
31     return 0;
32 }
```

Figure 1: A typical network program using asynchronous I/O and completion port on Windows. The pattern is also widely available on other platforms, such as Linux aio (`io_getevents` etc.), Solaris event completion (`port_get` etc.), and FreeBSD kqueue.

line 12), enqueues an asynchronous I/O request (`ReadFileEx`, line 16), and a worker thread waits on the I/O port for the completion of the I/O request (`GetQueuedCompletionStatus`, line 25). Similar interfaces are provided on other operating systems such as Linux (`aio`), Solaris (`port`), and BSD (`kqueue`), and are used by popular software such as the lighttpd web server that powers YouTube and Wikipedia.

The first challenge a developer must address is what calls are part of the interface that will be recorded and replayed. For example, in Figure 1, a developer might choose `socket` but not `ReadFileEx`. However, since during replay the call to `socket` is not executed, the returned socket descriptor is simply read from the recorded log rather than created. So the choice may crash `ReadFileEx` during replay and fail the application; the developer should choose both functions, or a lower layer that `ReadFileEx` uses. Section 2.2 formulates a number of rules that can guide the developer.

R2 generates stubs for the functions that the developer chooses to record and replay, and arranges that invocations to these functions will be directed to the corresponding stubs. To avoid reimplementing or modifying the implementation of the interposed interface, R2's goal is for the stubs to call the original intercepted functions and to record their results. This approach also allows R2 to record and replay functions for which only the binary versions are available.

To achieve this implementation goal and to ensure faithful replay, the stubs must address a number of implementation challenges. Consider the case in which the developer selects the functions from the Windows API (e.g., `GetQueuedCompletionStatus`, `ReadFileEx`, etc.) as the interface to be interposed. During a record run, the stubs must record in a log the socket descriptor and the completion port as integers, the output of `GetQueuedCompletionStatus` (e.g., the value of `cb` at line 26 and the content of `cb->buf` at line 27), along with other necessary information, such as the timestamp when the operating system starts `WorkerThread` as an upcall (callback) via a new thread.

During a replay run, the stubs will not invoke the intercepted functions such as `ReadFileEx` or `GetQueuedCompletionStatus`, but instead will read the results such as descriptors, the value of `cb`, and the content for `cb->buf` from the log. The stubs must also cause the memory side effects to happen (e.g., copying content into `cb->buf`). Finally, the replay run must also deliver upcalls (e.g., `WorkerThread`) at the recorded timestamps.

These requirements raise the following challenges:

- Use of intercepted functions by the implementation of the interposed interface. For example, the implementation itself may invoke the function `socket` and those invocations should not be recorded.

- Functions that have side effects. For example, to record and replay `ReadFileEx`, the stubs must record the content of `cb->buf` and fill it during replay. The stub for `ReadFileEx` must know that the second argument has side effects.

- Addresses returned by `malloc` must be identical during recording and replay. The code in Figure 1 requires that the value that `cb` receives at line 13 must *not* change from record to replay: the replay run reads the value of `cb` from the log at line 26 and that should be equal to the value returned by `malloc` at line 13; a different value for `cb` may lead to a crash in further uses (line 27 and 28).

- Threads created by the implementation of the interposed interface. The operating system, for example, might create threads to deliver events to the application, or might create "anonymous" threads to perform household tasks. The former should be recreated during replay, but the latter not.

- Execution order. Dependencies during recording must be preserved during replay. For example, `ReadFileEx`'s start of an asynchronous I/O must happen before the completion of that I/O event.

2.2 Choosing an Interface

As a starting point for choosing an interface, the developer must choose functions that form a *cut* in the call graph. Consider the call graph in Figure 2. The function `main` calls two functions `f1` and `f2`, and those two both call a third function `f3`, which interacts with the application's environment. The developer cannot choose to have only `f1` (or only `f2`) be the interposed interface. In the case of choosing `f1`, the effects of interactions by `f3` will be recorded only when called by `f1` but not by `f2`. During replay, `f3` interactions caused by `f1` will be read from the log but calls to `f3` from `f2` will be re-executed; `f1` and `f2` will see different interactions during replay.

For faithful replay, the interposed interface must form a cut in the call graph, thus the interface can be `f3` (cut 4) or both `f1` and `f2` (cut 1). Cuts 3 and 4 are also fine, but require R2 to track if `f3` was called from its side of the cut or from the other side. In the case of cut 3, R2 will not record invocations of `f3` by `f2` because `f2`'s invocations will be recorded and replayed. R2 supports all four cuts.

When the interposed interface forms a cut in the call graph, every function is either above the interface or below the interface. For example, if the developer chooses `f1` and `f2` as the interposed interface, then `main` is above and `f3` is below the interface. Functions above the interposed interface will be executed during replay, while functions below the interposed interface will not be executed during replay.

To ensure faithful replay, the cut must additionally satisfy two rules.

RULE 1 (ISOLATION) *All instances of unrecorded reads and writes to a variable should be either be below or above the interposed interface.*

Following the isolation rule will eliminate any shared state between code above and below the interface. A variable below the interface will be unobservable to functions above the interface; it is outside of the debugging scope of a developer. A variable above the interface will be faithfully replayed, executing *all* operations on it. Violating the isolation rule will result in unfaithful replay, because changes to a variable made by functions below the interface will not happen during the replay.

For the Windows API, the isolation rule typically holds. For example, all the operations on a file descriptor are performed through functions rather than direct memory access. During recording R2 records the file descriptor as an integer. During replay, R2 retrieves the integer from the log and returns it to functions above the interposed interface, without invoking the operating system to allocate descriptors. As long as R2 intercepts the complete set of file functions, the recorded file descriptor works correctly with the replayed application and ensures replay faithfulness.

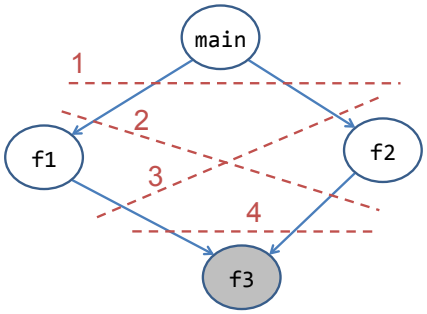


Figure 2: Four cuts in a call graph for record and replay. The function `f3` interacts with the environment.

RULE 2 (NONDETERMINISM) *Any source of nondeterminism should be below the interposed interface.*

If any nondeterminism is below the interposed interface, the impact to functions above the interface will be captured and returned to them. Violating this rule will result in unfaithful replay, because the behavior during replay will be different from during recording.

The sources of nondeterminism are as follows.

1. Calls that receive input data from the external (e.g., environment variables, files, and network).
2. Interprocess communications through shared memory (e.g., `WriteConsole` in Windows communicates with the CSRSS system service for standard input/output through a shared-memory segment).
3. Interactions between threads through shared variables (e.g., spinlocks).

R2 can handle the first source easily if the developer follows the isolation rule because input data from the external is received through functions. For the Windows API the developer must mark these functions being part of the interposed interface, which eliminates the nondeterminism.

For the second source, R2 can re-execute during replay if the replayed application only reads from shared memory. For more general cases the developer must mark the higher-level function that encloses the nondeterminism of shared memory accesses as being part of the interposed interface (e.g., `WriteConsole`).

The third source of nondeterminism stems from variables that are shared between threads via direct memory access instructions rather than functions. A similar

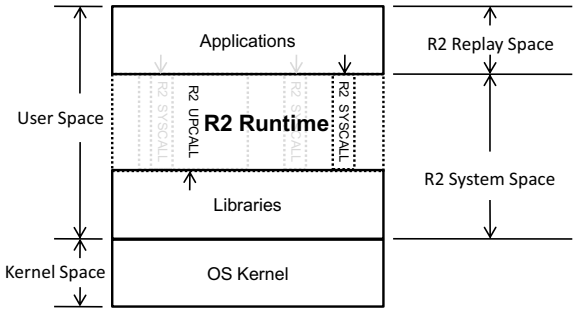


Figure 3: R2 overview. The user space is split into two spaces: R2 runtime that intercepts syscalls and underlying libraries are running in R2 system space; the application executes in R2 replay space.

case is the `rdtsc` instruction on the x86 architecture that reads the CPU timestamp counter. Often these instructions are enclosed by higher-level interface functions (e.g., lock and unlock of spinlocks). Developers must annotate them as being part of the interposed interface.

In practice, library APIs are good candidates for the interposed interface. First, library functions usually have variables shared between internal library routines and it is difficult to select only a subset of them as the interposed interface. Second, most sources of nondeterminism are contained within software libraries (e.g., spinlock variables in a lock library), and they affect external state only via library interfaces.

2.3 Isolation

R2 must address the implementation challenges listed in Section 2.1 for the stubs for the functions that are part of the appropriately-chosen interface. A starting point to handle these challenges is to separate the application that is being recorded and replayed from the code below the interface.

Inspired by isolation between kernel space and user space in operating systems, R2 defines two spaces (see Figure 3): **replay space** and **system space**. Unlike operating systems, however, the developer can decide which interface is the boundary between replay and system space. Like in operating systems, we refer to the functions in the interposed interface as **syscalls** (unless explicitly specified, all syscalls mentioned below are R2 syscalls instead of OS syscalls). Syscalls may register callback functions, which we call **upcalls**, that are issued later into replay space by system space runtime. With these terminologies, we can describe the spaces as follows:

- Replay space. All the code and data that is above the chosen syscall interface.
- System space. The R2 library and the underlying libraries, as well as any application code and data that is below the chosen syscall interface.

R2 records the output of syscalls invoked from application space, the input of upcalls invoked from system space, and their ordering. It faithfully replays them during the replay. R2 does not record and replay events in system space.

Consider the code in Figure 1 again. The developer may have chosen `malloc`, `socket`, `ReadFileEx`, `CreateThread`, `CreateIoCompletionPort`, and `GetQueuedCompletionStatus` as syscalls. The execution of `main` and `WorkerThread` is in replay space and the execution of the syscalls and the underlying libraries is in system space.

To record and replay syscalls and upcalls, R2 generates stubs from their function prototypes. R2 uses Detours [15] to intercept syscalls and upcalls, and detour their execution to the generated stub. For syscalls that take a function as an argument, R2 dynamically generates a stub for the function and passes on the address of the upcall stub to the system layer. This way when later the system layer invokes the upcall, it will invoke the upcall stub.

For syscalls that return data through a pointer argument, R2 must record the data that is returned during recording and copy that data into application space during replay. To do so correctly, the developer must annotate pointer arguments so that R2 knows what data should be recorded and how the stubs must transfer data across the syscall interface. Section 3 describes those annotations.

R2 maintains a replay/system mode bit for each thread to indicate whether the current thread is executing in replay or system space (analog to user/kernel mode bit). When the application in replay space invokes a syscall, the syscall stub sets the replay/system mode bit to system space mode, invokes the syscall, records its results, and restores the mode bit. Similarly, an upcall stub records the arguments, sets the mode bit to replay space mode, invokes the upcall, and restores the mode bit after the upcall returns.

This bit allows R2 to handle a call from system space to a function that is a syscall; if a syscall is called from system space then it must be executed without recording anything (e.g., a call to `socket` from system space). Similarly, if a syscall is called from system space and it has a function argument, then R2 will not generate an upcall stub for that argument. It also allows R2 to apply different policies to different spaces (e.g., allocate memory in a separate pool for code in replay space).

For functions that have state that straddles the boundary between system and replay space (e.g., `errno` in libc), the developer may be able to adjust the interface

to avoid such state (see Section 2.2) or may be able to duplicate the state by linking a static library (e.g., libc) in each space.

2.4 Execution Control

The separation in replay space and system space allows R2 to handle anonymous threads and threads that, for example, the operating system creates to deliver events to the application.

R2 starts as follows. When a user invokes R2 with the application to be recorded, R2's initial thread starts in system space. It loads the application's executable and treats the main entry as an upcall (i.e., the `main` function is turned into an upcall by generating an upcall stub). The stub sets the replay/system mode bit of the current thread to replay space mode, and invokes `main`. R2 assigns the thread a deterministic tag, which the stubs will also record. By this means, R2 puts the functions in the call graph starting from `main` till the syscall interface into replay space.

Anonymous threads that do not interact with the application will be excluded from replay. These threads will not call syscalls and upcalls and thus do not generate log entries during recording and are not replayed. However, if a thread started by the operating system performs an upcall (e.g., to trigger an application registered Ctrl-C handler on Windows), then the upcall stub will set the mode bit; the thread will enter replay space, and its actions will be recorded.

During replay, R2 will replay this upcall, but the thread for the upcall may not exist during replay. R2 solves this problem by creating threads on demand. Before invoking an upcall, R2 will first look up if the current thread is the one that ran the upcall during recording (by comparing the deterministic tag assigned by R2). If not, R2 will create the thread.

For faithful replay, R2 must replay all syscalls and upcalls in the same order as during a record run. In multithreaded programs (and single-threaded programs with asynchronous I/O) there may be dependencies between syscalls and upcalls. Section 4 introduces a few annotations that allow R2 to preserve a correct order.

2.5 Memory Management

If a developer chooses `malloc` and `free` as syscalls, R2 must ensure that addresses returned by `malloc` during recording are also returned during replay. If the addresses returned by `malloc` during replay are different, then during replay the bug that the developer is tracking down might not show up (e.g., invalid value in `cb->buf` will not be reproduced if a different `cb` is returned in Figure 1 because the program crashes before it reaches the buggy state). But, during replay, functions in system space that called `malloc` during recording will not be called during replay, and so `malloc` during replay is likely to return a different value.

To ensure faithful replay application must have an identical trace of `malloc`/`free` invocations to ensure that addresses during recording and replay are the same. R2 uses separate memory allocators for replay and system space. A call to `malloc` allocates memory from a dedicated pool if it is called in replay space (i.e., the mode bit of the current thread is of replay space mode), while it delegates the call to the original libc implementation if it is called in system space.

Memory addresses returned in system space may change due to inherent differences between record and replay, but those addresses are not observable in replay space so they will not impose any problems during replay.

A challenge is addresses allocated in system space but returned to replay space. For example, a syscall to `getcwd(NULL, 0)` to get working directory pathname may call `malloc` internally to allocate memory in system space and return its address to replay space. To ensure replay faithfulness R2 allocates a shadow copy in the dedicated pool for replay space and returns it to the application instead. R2 uses the annotation *xpointer* described in Section 3 to annotate such functions.

Similar to Jockey [28], threads that may execute in replay space have an extra stack allocated from the replay's memory pool, and R2 switches the two stacks on an upcall or syscall. This ensures that the memory addresses of local variables are the same during recording and replay.

Like all other library-based replay tools, R2 does not protect against a stray pointer in the application with which the application accidentally overwrites memory in system space. Such pointers are usually exposed and fixed early in the development cycle.

Resources other than memory (e.g., files, sockets) do not pose the same challenges as memory, as long as the developer has chosen the interposed interface well. R2 does not have to allocate these resources during replay, because the execution in replay space will touch these resources only via syscalls, which R2 records. Memory in contrast is changed by machine instructions, which R2 cannot record.

2.6 Stubs, Slots and Code Templates

R2 generates stubs from code templates. We have developed code templates for recording, replay, etc., but developers can add code templates for other operations that they would like stubs to perform. To allow for this extensibility, a stub is made of a number of slots, with each slot containing a function that performs a specific operation. For example, there is a slot for recording, one for replay, etc.

```
1 int recv (
2     [in] SOCKET s,
3     [out, bsize(return)] char *buffer,
4     [in] int len,
5     [in] int flags );
```

(a) annotated syscall/upcall prototype

```
1 BEGIN_SLOT(record_<?=$f->name?>, <?=$f->name?>)
2     logger << <?=$f->name?>_signature
3            << current_thr_tag;
4     <?if(is_syscall($f)) {?>
5        logger << return_val;<?}?>
6     <?$direction = is_syscall($f) ? 'out':'in';?>
7     <?foreach($f->params as $p) {
8        if ($p->has($direction)) {
9         if ($p->has('bsize')) {?>
10           logger.write(<?=$f->name?>,
11                        <?=$p->val('bsize')?>);
12        <?} else {?>
13    logger << <?=$f->name?>;
14    <?}}}?>
15 END_SLOT
```

(b) record slot function template

```
1 BEGIN_SLOT(record_recv, recv)
2     logger << recv_signature << current_thr_tag;
3     logger << return_val;
4     logger.write(buffer, return_val);
5 END_SLOT
```

(c) generated record slot function

Figure 4: Templates (in PHP [2]) and Slots. R2 uses record (and others like replay) code templates (e.g., (b)) to generate corresponding slot functions (e.g., (c)).

Figure 4 provides an overview of how a record slot function is generated for the `recv` syscall. Developers annotate the prototype of `recv` with keywords from Table 1; for `recv` this step will result in the prototype in Figure 4(a). (On Windows the developer does not have to do any annotation for `recv`, because R2 reuses the SAL annotations.) R2 uses a record template (see Figure 4(b)) to process the annotated prototype and produces the record slot function (see Figure 4(c)).

3 Data Transfers

R2 provides a set of keywords to define the data transfer at syscall and upcalls boundaries. These keywords help R2 isolate the replay and system space. This section presents the data transfer annotations and discusses how R2 uses them to ensure replay faithfulness.

3.1 Annotations

The annotations for data transfers fall into the following three categories.

Direction annotations define the source and destination of a data transfer. In Figure 4, keyword *in* on `s` and `len` indicates that they are read-only and transfer data into function `recv`, while *out* on `buffer` indicates that `recv` fills the memory region at `buffer` and transfers data out of the function. The return value of a function is implicitly annotated as *out*.

```
1  BOOL
2  [prepare(lpOverlapped, lpBuffer)]
3  ReadFileEx (
4      [in] HANDLE hFile,
5      [out] LPVOID lpBuffer,
6      [in] DWORD nNumberOfBytesToRead,
7      [in] LPOVERLAPPED lpOverlapped,
8      [in, callback]
9      LPOVERLAPPED_COMPLETION_ROUTINE completionCb);
10
11 typedef void
12 [commit(lpOverlapped, cbTransferred)]
13 (* FileIOCompletionRoutine) (
14     [in] DWORD dwErrorCode,
15     [in] DWORD cbTransferred,
16     [in] LPOVERLAPPED lpOverlapped );
17
18 BOOL
19 [commit(lpOverlapped, cbTransferred)]
20 GetOverlappedResult(
21     [in] HANDLE hFile,
22     [in] LPOVERLAPPED lpOverlapped,
23     [out] LPDWORD cbTransferred,
24     [in] BOOL bWait );
```

Figure 5: Asynchrony annotations: *prepare* indicates that `ReadFileEx` issues an asynchronous I/O request keyed by `lpOverlapped`, the completion of which is notified as either `FileIoCompletionRoutine` or `GetOverlappedResult`; *commit* indicates the request keyed by `lpOverlapped` is completed and the transferred data size is `cbTransferred`.

Buffer annotations define how R2 should serialize and deserialize data being transferred for record and replay. For `buffer` in a contiguous memory region in Figure 4, which is frequently seen in systems code, keyword *bsize* specifies the size, (e.g., *bsize*(*return*)), so that R2 can then automatically serialize and deserialize the buffer. For other irregular data structures such as linked lists (e.g., `struct hostent`, the return type of `gethostbyname`), R2 requires developers to provide customized serialization and deserialization via operator overloading on streams, which is a common C++ idiom.

Asynchrony annotations define asynchronous data transfers that finish in two calls, rather than in one. For example, in Figure 5 `ReadFileEx` issues an asynchronous I/O request keyed by `lpOverlapped`, which we call a **request key**. Developers use keyword *prepare* to annotate the syscall with the request key and the associated buffer `lpBuffer`. The completion of the request will be notified via either an upcall to `FileIoCompletionRoutine` (line 13) or a syscall to `GetOverlappedResult` (line 20), when the associated buffer has been filled in system space. In either case, developers use keyword *commit* to annotate it with the request key and transferred buffer size `cbTransferred`. R2 can then match `lpBuffer` with its size `cbTransferred` via the request key for record

and replay.

As mentioned in Section 2.5, some syscalls allocate a buffer in system space and the application may use the buffer in replay space. R2 provides the keyword *xpointer* to annotate this buffer, and will allocate a shadow buffer in replay space for the application, at both record and replay time. Data are copied to the shadow buffer from the real buffer in system space during recording, and from logs during replay. While data copy may add some overhead during recording, this kind of syscalls is infrequently used in practice.

Most such syscalls allocate new buffers locally and usually have paired "free" syscalls (e.g., `getcwd` and `free`, `getaddrinfo` and `freeaddrinfo`). Some others without paired "free" functions may return thread-specific or global data, such as `gethostbyname` and `GetCommandLine`. They should be annotated with *xpointer*(*thread*) and *xpointer*(*process*), respectively.

3.2 Code Generation

Figure 4 illustrates the record slot code template and the final record slot function for `recv`. The record slot functions log all the data transmitted from R2 system space to R2 replay space. The slot template (Figure 4(b)) generates code for recording the return value only when processing R2 syscalls (line 4). When scanning the parameters, it will record the data transfer according to the event type and annotated direction keywords (line 6 and 8). Specifically for upcalls, the input parameters and upcall function pointers are recorded so that R2 can execute the same callback with the same parameters (including memory pointers) during replay.

For prototypes annotated with *prepare*, the record slot template will skip recording the buffer. Instead, R2 uses another slot template to generate two extra record and replay slots for each prototype. One is for recording the buffer (including the buffer pointers), and the other is for replaying the buffer, which reads the recorded buffer pointers and fills them with the recorded data. These two slots will be plugged into stubs for prototypes labeled with *commit* at the record and replay phases, respectively. This approach ensures that the memory addresses during replay are identical to the ones returned during recording for asynchronous I/O operations.

4 Execution Orders

For faithful replay, R2 must replay all syscalls and upcalls in the same order as during a record run. For single-threaded programs asynchronous I/O raises some challenges. For multithreaded programs that run on multiprocessors, recording the right order is more challenging, because syscalls and upcalls can happen concurrently, but dependencies between syscalls and upcalls executed by different threads must be maintained. R2 provides developers with two annotations to express such dependencies. This section describes how R2 handles the recording and replay execution order.

```
1  BOOL
2  [sync(hMutex)]
3  ReleaseMutex (
4      [in] HANDLE hMutex );
5
6  DWORD
7  [sync(hMutex)]
8  WaitForSingleObject (
9      [in] HANDLE hMutex,
10     [in] DWORD dwMilliseconds );
```

Figure 6: Syscall-syscall causality annotations using *sync*. R2 serializes the syscall events with the same sync key `hMutex` to obtain an event order, and the causalities between these events are implicitly held by the event sequence.

4.1 Event Definition

In R2 there are three events: syscalls, upcalls, and causalities. R2 uses the causality events to enforce the happens-before relation between events executed by different threads. Consider the following scenario: one thread uses a syscall to put an object in a queue, and later a second thread uses another syscall to retrieve it from the queue. During replay the first syscall must always happen before the second one; otherwise, the second syscall will receive an incorrect result. Using annotations, these causalities are captured in **causality events**. A causality event has a source event e_1 and a destination event e_2, which captures $e_1 \prec e_2$. R2 generates a slot function that it stores in both e_1 and e_2's stubs, which will cause the causality event to be replayed when R2 replays e_1 and e_2.

R2 captures two types of causality events:

- syscall-syscall: a syscall depends on an earlier one, e.g., signal and wait;
- syscall-upcall: a syscall registers a callback that is executed later as an upcall.

To capture syscall-syscall causality, R2 provides the keyword *sync* to annotate syscalls that operate on the same resource. Figure 6 presents an example, where a call to `WaitForSingleObject` that acquires a mutex depends on an earlier call to `ReleaseMutex` that releases it. Developers can then annotate them with *sync*(`hMutex`). We call the mutex `hMutex` a **sync key**. R2 creates causality events for syscalls with the same sync key. In addition to mutexes, a sync key can be any expression that refers to a unique resource. For asynchronous I/O operations (see Figure 5), R2 uses `lpOverlapped` as the sync key.

```
1 HANDLE CreateThread (
2     [in] LPSECURITY_ATTRIBUTES lpThreadAttributes,
3     [in] SIZE_T dwStackSize,
4     [in, callback] LPTHREAD_START_ROUTINE lpStartCb,
5     [in] LPVOID lpParameter,
6     [in] DWORD dwCreationFlags,
7     [out] LPDWORD lpThreadId );
```

Figure 7: A syscall-upcall causality annotation using *callback*. R2 converts the callback argument `lpStartCb` into an upcall stub; when the upcall is delivered, the syscall-upcall causality will be captured.

```
1:  ▷ c(t_0) ← 0
2:  procedure UPDATECLOCKUPONEVENT(e)
3:      if e.type = CAUSALITY_EVENT then
4:          c(t) ← max(c(e.source), c(t)) + 1
5:          c(e) ← c(t)
6:      else
7:          c(t) ← c(t) + 1
8:          c(e) ← c(t)
9:      end if
10: end procedure
```

Figure 8: Algorithm for calculating event clocks. The procedure is invoked when processing every event.

For syscall-upcall causality, developers can use the keyword *callback* to mark the dependency, as illustrated in Figure 7. R2 generates a causality event for the syscall to `CreateThread` and the upcall to `lpStartCb`.

4.2 Recording Event Order

R2 uses a Lamport clock [18] to timestamp all events and uses that timestamp to replay. It assigns each thread t a clock $c(t)$ and each event e a clock $c(e)$. Figure 8 shows R2's algorithm to calculate the Lamport clock for each event.

During recording, R2 first sets the clock of the main thread to 0 (line 1). Then, when an event is invoked, R2 calculates the clock for that event using the procedure UPDATECLOCKUPONEVENT. For non-causality events, R2 simply increases the current thread's clock by one (line 7) and then assigns that value to the event (line 8).

For a causality event, R2 updates the current thread's clock to the greater value of itself and the clock from the source event of the causality (note that $e \leftarrow e.source \prec e.destination$), and increases it by one (line 4). R2 also assigns this value to the causality event (line 5).

When a thread invokes the destination event of a causality event, R2 first runs the slot function for the causality event, which invokes UPDATECLOCKUPONEVENT with the causality event as argument. This will cause the clock of the causality event to be propagated to the thread (line 7 in Figure 8).

There are several possible execution orders that preserve the happens-before relation, as we discuss next.

4.3 Replaying Event Order

R2 can use two different orders to record and replay events: total-order and causal-order. Total-order execution can faithfully replay the application, but may slow down multithreaded programs running on a multiprocessor, and may hide concurrency bugs. Causal-order execution allows true concurrent execution, but may replay incorrectly if the program has race conditions.

4.3.1 Total-Order Execution

Like liblog [10], in total-order execution mode, R2 uses a token to enforce a total order in replay space, including execution slices that potentially could be executed concurrently by different threads. During recording when a thread enters replay space (i.e., returning from a syscall or invoking an upcall), it must acquire the token first and calculate a timestamp if an upcall is present. When a thread leaves replay space (i.e., invoking a syscall), R2 assigns a timestamp to the syscall and then releases the token. On a token ownership switch R2 generates a causality event to record that the token is passed from one thread to another. This design serializes execution in replay space during recording, although threads executing in system space remain concurrent. The result is a total order on all events.

During replay, R2 replays in the recorded total order. As it replays the events, R2 will dynamically create new threads for events executed by different threads during recording (as described earlier in Section 2.4). It will ensure that these threads execute in the same order as enforced by the token during recording. The reason to use multiple threads, even though the execution in replay space has been serialized, is that developers may want to pause a replay and use a standard debugger to inspect the local variables of a particular thread to understand how the program reached the state it is in. In addition, using multiple threads during replay ensures that thread-specific storage works correctly.

4.3.2 Causal-Order Execution

Causal-order execution, however, allows threads to execute in parallel in both replay and system space. R2 does not impose a total order in replay space, it just captures the causalities of syscall-syscall and syscall-upcall. Therefore the application will achieve the same speedup in causal-order execution.

To implement causal-order execution, R2 reuses the replay facility for total-order execution. R2 processes the causal-order event log before replay, uses a log converter to translate the event sequences into any total order that preserves all causalities, and replays using the total-order

execution. If the program has an unrecorded causality (e.g., data race), R2 cannot guarantee to replay these causalities faithfully in causal-order execution. We have not fully implemented the log converter yet, since our focus is replaying distributed applications and total-order execution has been good enough for this purpose.

5 Defining Your Own Syscalls

We have annotated a large set of Win32 API calls for R2 to support most Windows applications without any effort from developers, including those required to be annotated with *xpointer*, *prepare*, *commit*, and *sync*. Sometimes developers may want to define their own syscalls, either to enclose nondeterminism in syscalls (e.g., `rdtsc`, spin lock cases in Section 2.2) or to reduce recording overhead. In this section we use MPI and SQLite as examples to explain how to do this in general.

5.1 MPI

MPI is a communication protocol for programming parallel computing applications. An MPI library usually has nondeterminism that cannot be captured by intercepting Win32 functions (e.g., MPICH [4] uses shared-memory and spinlocks for interprocess communication). To replay MPI applications, this nondeterminism must be encapsulated by R2 syscalls. Therefore, we annotate all MPI functions as syscalls, making the entire MPI library run in system space. Since the MPI library is well encapsulated by these MPI functions, doing this ensures that both rules in Section 2.2 are satisfied.

Annotating MPI functions is an easy task. Most functions only require the *in* and *out* annotations at parameters. Several "non-blocking" MPI functions (e.g., `MPI_Irecv` and `MPI_Isend`) use asynchronous data transfer, which is easily captured using the *prepare* and *commit* annotations. Figure 9 shows the annotated `MPI_Irecv` and `MPI_Wait`, which issue an asynchronous receive and wait for the completion notification, respectively. These functions are associated based on the `request` parameter by the annotations. Section 8.2 presents the number of annotations needed.

5.2 SQLite

SQLite [3] is a widely-used SQL database library. A client accesses the database by invoking the SQLite API. Using Win32 level syscalls, R2 can faithfully replay SQLite client applications. Additionally, developers can add the SQLite API to R2 syscalls so that R2 will record the outputs of SQLite API, and avoid recording file operations issued by the SQLite library in system space.

In certain scenarios, recording at the SQLite API layer can dramatically reduce the log size, compared with recording at the Win32 layer. For example, some

```
1  int
2  [prepare(request, buf)]
3  MPI_Irecv (
4      [out, bsize(MPISize(type, count))] void *buf,
5      [in] int count,
6      [in] MPI_Datatype type,
7      [in] int source,
8      [in] int tag,
9      [in] MPI_Comm comm,
10     [out] MPI_Request *request );
11
12 int
13 [commit(request)]
14 MPI_Wait (
15     [in] MPI_Request *request,
16     [out] MPI_Status *status);
```

Figure 9: Example of asynchrony annotations on MPI functions. The size of the returned buffer at *commit* is by default the registered *bsize* at *prepare* in `MPI_Irecv`.

```
1  int sqlite3_prepare (
2      [in]  sqlite3 * db,
3      [in]  const char * zSql,
4      [in]  nByte,
5      [out] sqlite3_stmt ** ppStmt,
6      [out] const char ** pzTail );
7
8  int sqlite3_column_int (
9      [in] sqlite3_stmt * pStmt,
10     [in] iCol );
```

Figure 10: Example of annotated SQLite functions.

`SELECT` queries may scan a large table but return only a small portion of matched results. For these queries, recording only the final results is more efficient than recording all data fetched from database files. Section 8.4 shows the performance benefits of this approach.

Figure 10 shows two annotated SQLite functions: `sqlite3_prepare` and `sqlite3_column_int` are typical routines for compiling a query and retrieving column results, respectively.

6 Annotations for Optimization

In this section, we introduce two additional annotation keywords to optimize R2's performance.

Cache annotation. By inspecting logs we find that a few syscalls are invoked much more frequently than others—more than two orders of magnitude. Also, most of them return only a status code that does not change frequently (e.g., `GetLastError` on Windows returns zero in most cases). To improve recording performance R2 introduces keyword *cache* to annotate such syscalls. Every time a syscall annotated with *cache* returns a status code, R2 compares the value with the cached one from the same syscall; only when it changes will R2 record the new value in the log and update the cache. An Apache experiment in Section 8.5.1 shows that this optimization reduces the log size by a factor of 17.66%.

Manually Coded Modules	kloc
annotation parser & code generator	4.1
core (interception, isolation, slot)	1.3
upcall (*callback*)	0.7
causality (*sync*)	1.9
aio (*prepare*/*commit*)	1.3
record-replay (memory, data, event)	10.2
Total	19.5

Automatically Generated Modules	kloc
callback.Win32	3.6
causality.Win32	2.9
aio.Win32	1.9
R2.Win32	102.0
R2.app.specific	-
Total	110.2

Table 2: R2 modules.

Reproduce annotation. Some application data can be reproduced at replay time *without* recording. Consider a BitTorrent node that receives data from other peers and writes them to disk. It also reads the downloaded data from disk and sends them to other peers. It is safe to record all input data for replay, i.e., both receiving from network and reading from disk. However, R2 need not record the latter. Developers can use keyword *reproduce* to annotate file I/O syscalls in this case. R2 then generates stubs from a specific code template, to re-execute or simulate I/O operations, instead of recording and feeding. Network I/O, such as intra-group communications of an MPI application, can be reduced similarly [33]. Section 8.5.2 and 8.5.3 show that this optimization can reduce log sizes ranging from 13.7% to 99.4% for BitTorrent and MPI experiments.

7 Implementation

R2 is decomposed into a number of reusable modules. Table 2 lists each module and lines of code (loc). In sum, we have manually written 19 kloc; 110 kloc are generated automatically for R2's Win32 layer implementation.

We have annotated more than one thousand Win32 API calls (see statistics in Table 4). Although this well covers commonly-used ones, Win32 has a much wider interface, and we may still have missed some used by applications. Therefore, we have built an API checker that scans the application's import table and symbol file to detect missing API calls when an application starts.

During replay R2 may still fail because of some unrecorded non-determinisms, e.g., data races not enclosed by R2 syscalls. Since non-determinisms usually lead to different control flow choices thus different syscall invocation sequences, R2 records the syscall signature (e.g., name) and checks it during replay (check whether the current invoked syscall has the same signature with that from the log). By this means, R2 can efficiently detect the mismatch at the first time when R2 gains control after the deviation. When a mismatch is found, R2 reports the current Lamport clock and the mismatch. The developer can then replay the application again with a breakpoint set at the Lamport clock value minus one. When the breakpoint is hit, the developer can then examine how the program reached a different state during replay, and fix the problem (e.g., by adjusting the interposed interface). We have found that this approach works well to debug the R2 interface.

Category	Software Package
web server	Apache, lighttpd, Null HTTPd
database	SQLite, Berkeley DB, MySQL
distributed system	libtorrent, Nyx, PacificA
virtual machine	Lua, Parrot, Python
network client	cURL, PuTTY, Wget
misc.	zip, MPICH

Table 3: Software packages successfully replayed.

8 Evaluation

We have used R2 to successfully replay many real-world applications. Table 3 summarizes an incomplete list. Most of the applications are popular system applications, such as multi-threaded Apache and MySQL servers, which we believe R2 is the first to replay. Nyx [7] is a social network computation engine for MSN and PacificA [19] is a structured storage system similar to Google Bigtable [6], both of which are large, complex distributed systems and have used R2 for replay debugging. The implementation of these applications requires addressing the challenges mentioned in Section 2.

This section answers the following questions.

- How much effort is required to annotate the syscall-upcall interface?
- How important are annotations to successful replay of applications?
- How much does R2 slowdown applications during recording?
- How effective are custom syscall layers and annotations (*cache* and *reproduce*) in reducing log size and optimizing performance?

Similar to previous replay work, we do not evaluate the replay performance, because replay is usually an interactive process. However, the replayed application without any debugging interaction runs much faster

Interface	#func	*in*	*out*	*bsize*	*cb*	*xptr*	*pr*	*ci*	*sync*	*cache*	*reproduce*	#serial	kloc
Win32	1,301	1,100	631	168	53	11	17	4	30	2	3	7	110.2
MPI	191	171	150	20	6	4	6	4	1	0	4	5	22.2
SQLite	153	150	16	4	19	3	0	0	7	0	0	0	15.7

Table 4: Information about annotations and code generation. Columns with annotation keywords show the number of functions for each keyword. Keywords *callback*, *xpointer*, *prepare*, *commit*, are abbreviated to *cb*, *xpr*, *pr*, *ci*, respectively. The last two columns list the number of functions that require customized (de)-serialization, and lines of automatically generated code, respectively.

than when recording (e.g., a replay run of BitTorrent file downloading is 13x faster).

8.1 Experimental Setup

All experiments are conducted on machines with 2.0 GHz Xeon dual-core CPU, 4 GB memory, two 250 GB, 7200 /s disks, running Windows Server 2003 Service Pack 2, and interconnected via a 1 Gbps switch. Unless explicitly specified, the application data and R2 log files are kept on the same disk; the record run uses total-order execution; all optimizations (i.e., cache and reproduce) are turned off.

8.2 Annotation Effort

We annotated the first set (500+ functions) of the Win32 syscall interface within one person-week, and then annotated as needed. We reused *in*, *out*, *bsize*, and *callback* from the Windows Platform SDK, and manually added the other annotations (i.e., *xpointer*, *prepare*, *commit*, *sync*); we manually annotated only 62 functions.

We found that once we decided how to annotate a few functions for a particular programming concept (e.g., asynchronous I/O, or synchronization), then we could annotate the remaining functions quickly. For example, after we annotated the file-related asynchronous I/O functions, we quickly went through all the socket related asynchronous I/O functions.

For the two other syscall interfaces, MPI and SQLite (discussed in Section 5), we spent two person-days annotating each before R2 could replay MPI and SQLite applications. The first four keywords (*in*, *out*, *bsize*, and *callback*) are trivial and cost us little time, and we mainly spent our time on other annotations and writing customized (de)-serialization functions.

Table 4 lists for the three syscall interface the annotations used, how many functions used them, the number of functions that needed customized (de)-serialization, and the lines of code auto generated (approximately 148 kloc). The table shows that the annotations are important for R2; without them it would have been a tedious and error-prone job to manually write so many stubs.

Configuration	Request#/s	Slowdown	Log rate
native	1242.23	-	-
stub only	1241.75	0.04%	-
log	1125.58	1.34%	0.760
causal-order	1197.52	3.60%	1.114
total-order	1129.94	9.04%	0.781

Table 5: Apache performance under different R2 configurations (cache on). Log rate is measured in KB/req. Client concurrency level is 50 and the download file size is 64 KB.

8.3 Performance

We measure the recording performance of R2 using the Apache web server 2.2.4 with its default configuration (250 threads) and the standard ApacheBench client, which is included in the same package.

Table 5 shows the reduction in request throughput and the log overhead under different R2 configurations. We use ApacheBench to mimic 50 concurrent clients, all of which download 64 KB static files (which is a typical web page size). Each configuration in the table executes 500,000 requests. As we can see, the stub, the logger, and the causal-order execution have little performance impact; the total-order execution imposes a slowdown up to 9.04%, which we believe to be acceptable for the purpose of debugging. The log produced for each request is approximately 0.8 KB, slightly bigger for causal-order execution mode since it needs to log more causality events.

Figure 11 shows the results for total-order and causal-order configurations, with a varied number of concurrent clients and file size. We see that when the download file is larger, the slowdown is smaller. This is because the larger file size means that the execution in replay space costs less CPU time, and the slow down imposed by total order execution is less. For the smallest size of 16KB we tested, the average slowdown for all concurrency levels is 11.2% under total-order configuration and 4.9% under causal-order configuration.

In addition to Apache, we have also measured the performance of many other applications while recording. The slowdown for most cases is moderate (e.g., 9% on

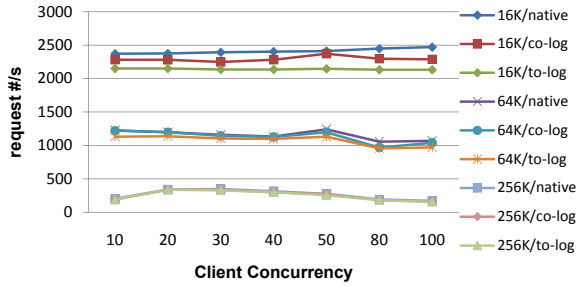


Figure 11: Apache performance using total-order and causal-order configurations, with a varied number of concurrent clients and file size.

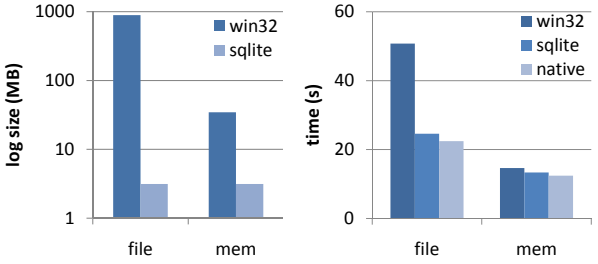


Figure 12: SQLite log size and execution time at Win32 and SQLite interfaces using FILE and MEM configurations.

average for the standard MySQL benchmark [12]). There are exceptions, such as the SQLite case below. The performance of these exceptions, however, can be improved using either customized syscall layer or optimization annotations.

8.4 Customized Syscall Layers

This section evaluates the performance of R2 for SQLite using two syscall layers, i.e., Win32 and SQLite, which was discussed in Section 5.2.

We adopted a benchmark from Nyx [7], which calculates the degree distribution of user connections in a social network graph. The calculation is expressed as a query: `SELECT COUNT(*) FROM edge GROUP BY src_uid`. We perform the query 10 times, and measure log size as well as execution time. The data set contains 156,068 edges and is stored in SQLite; the file size is approximately 3 MB.

By default SQLite stores temporary tables and indices in files; it can store them in memory by setting an option. We use FILE and MEM to name the two configurations. Other options are default values.

Figure 12 shows log size and execution time for recording at the two syscall layers, respectively. In either configuration, recording at the Win32 interface produces much larger logs (890 MB and 35 MB), compared to recording at the SQLite interface (only 3 MB). The slowdown factors of recording at the two interfaces are 126.3% and 9.6% under FILE, 17.8% and 7.3% under MEM, respectively.

Note that the recording at the SQLite interface produces the same size of log for the two configurations, because the SQL layer does not involve file I/O and the log size is not effected by the configurations.

From these results we can see that recording at the SQLite layer can reduce log overhead and improve performance, if for a query SQLite must perform I/O frequently.

Cached Syscalls	Call#	Miss#	Hit Ratio
`GetLastError`	618,015	99,948	83.82%
`CloseHandle`	150,016	2	99.99%
`setsockopt`	150,003	1	99.99%
`FindClose`	100,147	2	99.99%
`WaitForSingleObject`	100,014	4	99.99%
Total	1,118,195	99,957	91.06%

Table 6: Apache cached syscalls with cache miss and hit statistics. Client concurrency level is 50 and download file size is 64 KB.

8.5 Optimization Annotations

As discussed in Section 6, R2 introduces two annotation keywords to improve its performance. We evaluate them in this section.

8.5.1 The cache Annotation

We use the Apache benchmark again to evaluate the cache annotation. The experiment runs R2 in total-order execution mode. The client's concurrency level is 50 and the file downloaded is 64 KB. Profiling Apache shows that 5 out of 61 syscalls contribute more than 50% of syscall. We use the cache annotation for these syscalls. Table 6 shows how many times these syscalls were invoked and did not hit the cache in one test run. We see that the return values of these syscalls were mostly in the cache, and that the average hit ratio is 91.06%. This reduced the log size from 21.99 MB to 18.1 MB (approximately 17.66% reduction). We applied the cache optimization to only five syscalls, but we could gain more benefits if we annotated more syscalls.

8.5.2 Reproduced File I/O

As discussed in Section 6, when the reproduce annotation is used for file I/O when recording BitTorrent, the file content that is read from a disk is not recorded, and the related file syscalls are re-executed during replay.

We use a popular C++ BitTorrent implementation libtorrent [1] to measure the impact of this annotation. The experiment was conducted on 11 machines, with one

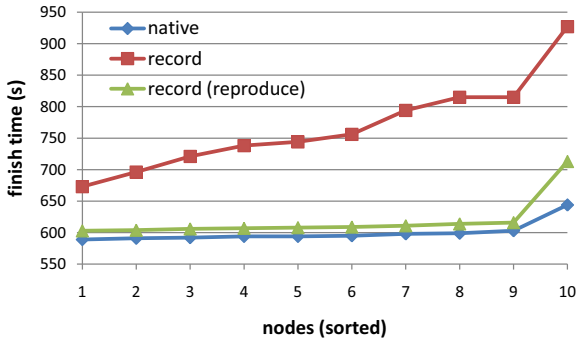


Figure 13: Finish time of 10 BitTorrent nodes in runs of native, record without and with reproduced I/O optimization.

seed and 10 downloaders. The seed file size was 4 GB; the upload bandwidth was limited to 8 MB/s. R2 ran in total-order mode with cache optimization off.

The average log sizes of the recording run without and with the reproduce annotation are 17.1 GB and 5.4 GB, respectively. The optimization reduces the log size by 68.2%. Relative to the 4 GB file size, the two cases introduce 297.5% and 26.4% log size overhead, respectively.

Figure 13 presents the finish time of the 10 downloaders for a native run, as well for recorded runs without and with the reproduce annotation. On average, the slowdown factors of recording without and with the annotation are 28% and 3%, respectively. We can see that the *reproduce* annotation is effective when recording I/O intensive applications, reducing both the log and performance overhead.

8.5.3 Reproduced Network I/O

We use the MPI syscall layer to evaluate the benefit of the *reproduce* annotation for network I/O. The experiment was conducted in our MPI-replay project [33], which uses R2. We annotated MPI functions using the *reproduce* annotation so that the messages are not recorded but reproduced during replay. Table 7 shows the effectiveness for two typical MPI benchmarks: GE [14] and PU [11]. We see that the client process of PU gains much benefit from this keyword; the log size is reduced by more than 99.4%. For GE, it also results in a log size reduction, but of about 13.7%.

	w/o opt (MB)		w/ opt (MB)		Ratio
	node 0	node 1	node 0	node 1	
GE	55.3	55.3	47.7	47.7	86.3%
PU	4.5	1170.0	5.7	1.7	0.6%

Table 7: Reproduced network I/O optimization on MPI. "Ratio" is the log size with this optimization compared that without. The other fields are R2 log size on each node.

9 Related Work

R2 borrows many techniques from previous replay tools, in particular from library-based ones. This section relates R2 to them in more detail.

Library-based replay. Several replay tools use a library-based approach. The closest work is Jockey [28] and liblog [10], where a runtime user-mode library is injected into a target application for record and replay. We borrow many ideas from these tools (e.g., using a token to ensure total-order execution) but extend the library-based approach to a wider range of applications using stricter isolation (inspired by operating system kernel ideas), by flexible customization of the record and replay interface, by annotations, and automatic generation of stubs.

R2 also isolates the application from the tool in a different way. For example, Jockey tries to guarantee that the application behaves the same with and without record and replay, and liblog shares the same goal. Consequently, they both send the memory requests from the *tool* to a dedicated memory region to avoid changing the memory footprint of the application. As discussed in Section 1, R2 aims for replay faithfulness instead, and it manages memory requests from the *application*.

Jockey and liblog have a fixed interface for record and replay (a mix of system and libc calls); any nondeterminism that is not covered will cause replay to fail. R2 enables developers to annotate such cases using keywords on functions of higher-level interfaces to enclose nondeterminism.

On the implementation side, both Jockey and liblog have manually implemented many stubs (100+); R2's more automatic approach makes it easier to support a wider range of syscalls. For example, Jockey does not support multithreading; liblog also does not support asynchronous I/O and other functions.

RecPlay [25] captures causalities among threads by tracking synchronization primitives. R2 uses that idea too, but also captures other causalities (e.g., syscall-upcall causalities). RecPlay uses a vector clock during replay and can detect data races. This feature could be useful to R2 too.

Another library-based approach but less related is Flashback [29], which modifies the kernel and records the input of the application at system call level. Since it is implemented as a kernel driver, it is less easy to deploy and use than R2.

Domain-specific replay. There are a large number of replay tools focusing on applications using restricted programming models, such as distributed shared mem-

ory [27] or MPI [26], or in specific programming languages such as Standard ML [30] or Java [17]. This approach is not suitable for the system applications that R2 targets. In fact, we built a replay tool before [21], which relies on the programmers to develop their applications using our own home-grown API. The limitation of this work propelled us to design and build R2.

Whole system replay. A direct way to support legacy applications is to replay the *whole* system, including the operating system and the target applications. A set of replay tools aim at this target, either using hardware support [32, 24, 23] or virtual machines [8, 16, 5]. They can replay almost every aspect of an application's environment faithfully, including scheduling decisions inside the operating system, which makes them suitable to debug problems such as race conditions. They can achieve similar performance as R2; ReVirt [8], for instance, has a slowdown of 8% for rebuilding the kernel or running SpecWeb99 benchmarks. However, they can be inconvenient and expensive to deploy. For example, developers must create a virtual machine and install a copy of the operating system to record and replay an application.

Annotations. Annotations on functions are widely used in many fields. For example, a project inside Microsoft uses SAL and static analysis to find buffer overflows [13]. Instead, SafeDrive [34] inserts runtime checks where static analysis is insufficient according to the annotations. While they all focus on finding bugs, R2 uses annotations to understand function side effects, and generates code to record and replay them.

Enforcing isolation with binary instrumentation. XFI [9] is a protection system which uses a combination of static analysis with inline software *guards* that perform checks at runtime. It ensures memory isolation by introducing external checking modules to check suspicious memory accesses at runtime. Because XFI monitors the memory access at instruction level, its overhead varies from 5% to a factor of two, depending on how the static analysis works and also the benchmark. R2 isolates at function interfaces and targets replay, which allows it to be more loose in its isolation in some ways (i.e., it does not have to protect against attacks), but more strict in other ways (i.e., memory addresses cannot change from recording to replay).

10 Conclusion

R2 uses kernel ideas to split an application's address space into a replay and a system space, allowing strict separation between the application and the replay tool. With help from the developer, who specifies some annotations on the syscall interface, R2 carefully manages transitions between replay and system space at the syscall interface, and isolates resources (e.g., threads and memory) within a space.

The annotations also allow R2 to generate syscall and upcall stubs from code templates automatically, and make it easy for developers to choose different syscall/upcall interfaces (e.g., MPI or SQLite). It also allows developers to enclose nondeterminism and avoid shared state between replay and system space. Annotations for optimizations can reduce the record log size and improve performance.

By using these ideas R2 extends recording and replay to applications that state-of-the-art library-based replay tools cannot handle. R2 has become an important tool for debugging applications, especially distributed ones, and a building block for other debugging tools, such as runtime hang cure [31], distributed predicate checking [20], task hierarchy inference [22], and model checking.

Acknowledgment

We thank Alvin Cheung, Evan Jones, John McCullough, Robert Morris, Stefan Savage, Alex Snoeren, Geoffrey Voelker, our shepherd, David Lie, and the anonymous reviewers for their insightful comments. Thanks to our colleagues Matthew Callcut, Tracy Chen, Ruini Xue, and Lidong Zhou for valuable feedback.

References

[1] libtorrent 0.11. `http://libtorrent.sourceforge.net/`.

[2] PHP: Hypertext preprocessor. `http://www.php.net/`.

[3] SQLite 3.5.8. `http://www.sqlite.org/`.

[4] D. Ashton and J. Krishna. *MPICH2 Windows Development Guide*. Argonne National Laboratory, 2008.

[5] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE*, 2006.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[7] Y. Chen, T. Chen, M. Chen, and Z. Zhang. Islands in the MSN Messenger buddy network. In *SocialNets*, 2008.

[8] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, 2002.

[9] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *OSDI*, 2006.

[10] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *USENIX*, 2006.

[11] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[12] Z. Guo, X. Wang, X. Liu, W. Lin, and Z. Zhang. Towards pragmatic library-based replay. Technical Report MSR-TR-2008-02, Microsoft Research, 2008.

[13] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *ICSE*, 2006.

[14] Z. Huang, M. K. Purvis, and P. Werstein. Performance evaluation of view-oriented parallel programming. In *ICPP*, 2005.

[15] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *USENIX Windows NT Symposium*, 1999.

[16] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX*, 2005.

[17] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed Java applications. In *IPDPS*, 2000.

[18] L. Lamport. Time, clocks and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

[19] W. Lin, M. Yang, L. Zhang, and L. Zhou. PacificA: Replication in log-based distributed storage systems. Technical Report MSR-TR-2008-25, Microsoft Research, 2008.

[20] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D^3S: Debugging deployed distributed systems. In *NSDI*, 2008.

[21] X. Liu, W. Lin, A. Pan, and Z. Zhang. WiDS checker: Combating bugs in distributed systems. In *NSDI*, 2007.

[22] H. Mai, C. Gao, X. Liu, X. Wang, and G. M. Voelker. Towards automatic inference of task hierarchies in complex systems. In *HotDep*, 2008.

[23] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using Strata. In *ASPLOS*, 2006.

[24] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *ISCA*, 2005.

[25] M. Ronsse and K. D. Bosschere. RecPlay: A fully integrated practical record/replay system. *TOCS*, 17(2):133–152, 1999.

[26] M. Ronsse, K. D. Bosschere, and J. C. de Kergommeaux. Execution replay for an MPI-based multi-threaded runtime system. In *ParCo*, 1999.

[27] M. Ronsse and W. Zwaenepoel. Execution replay for treadmarks. In *PDP*, 1997.

[28] Y. Saito. Jockey: A userspace library for record-replay debugging. In *AADEBUG*, 2005.

[29] S. Srinivasan, C. Andrews, S. Kandula, and Y. Zhou. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *USENIX*, 2004.

[30] A. Tolmach and A. W. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, 1995.

[31] X. Wang, Z. Guo, X. Liu, Z. Xu, H. Lin, X. Wang, and Z. Zhang. Hang analysis: Fighting responsiveness bugs. In *EuroSys*, 2008.

[32] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.

[33] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. M. Voelker. MPIWiz: Subgroup reproducible replay of MPI applications. In *PPoPP*, 2009.

[34] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *OSDI*, 2006.

KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

Cristian Cadar, Daniel Dunbar, Dawson Engler *
Stanford University

Abstract

We present a new symbolic execution tool, KLEE, capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs. We used KLEE to thoroughly check all 89 stand-alone programs in the GNU COREUTILS utility suite, which form the core user-level environment installed on millions of Unix systems, and arguably are the single most heavily tested set of open-source programs in existence. KLEE-generated tests achieve high line coverage — on average over 90% per tool (median: over 94%) — and significantly beat the coverage of the developers' own hand-written test suite. When we did the same for 75 equivalent tools in the BUSYBOX embedded system suite, results were even better, including 100% coverage on 31 of them.

We also used KLEE as a bug finding tool, applying it to 452 applications (over 430K total lines of code), where it found 56 serious bugs, including three in COREUTILS that had been missed for over 15 years. Finally, we used KLEE to crosscheck purportedly identical BUSYBOX and COREUTILS utilities, finding functional correctness errors and a myriad of inconsistencies.

1 Introduction

Many classes of errors, such as functional correctness bugs, are difficult to find without executing a piece of code. The importance of such testing — combined with the difficulty and poor performance of random and manual approaches — has led to much recent work in using *symbolic execution* to automatically generate test inputs [11, 14–16, 20–22, 24, 26, 27, 36]. At a high-level, these tools use variations on the following idea: Instead of running code on manually- or randomly-constructed input, they run it on symbolic input initially allowed to be "anything." They substitute program inputs with symbolic values and replace corresponding concrete program operations with ones that manipulate symbolic values. When program execution branches based on a symbolic value, the system (conceptually) follows both branches, on each path maintaining a set of constraints called the *path condition* which must hold on execution of that path. When a path terminates or hits a bug, a test case can be generated by solving the current path condition for concrete values. Assuming deterministic code, feeding this concrete input to a raw, unmodified version of the checked code will make it follow the same path and hit the same bug.

Results are promising. However, while researchers have shown such tools can sometimes get good coverage and find bugs on a small number of programs, it has been an open question whether the approach has any hope of consistently achieving high coverage on real applications. Two common concerns are (1) the exponential number of paths through code and (2) the challenges in handling code that interacts with its surrounding environment, such as the operating system, the network, or the user (colloquially: "the environment problem"). Neither concern has been much helped by the fact that most past work, including ours, has usually reported results on a limited set of hand-picked benchmarks and typically has not included any coverage numbers.

This paper makes two contributions. First, we present a new symbolic execution tool, KLEE, which we designed for robust, deep checking of a broad range of applications, leveraging several years of lessons from our previous tool, EXE [16]. KLEE employs a variety of constraint solving optimizations, represents program states compactly, and uses search heuristics to get high code coverage. Additionally, it uses a simple and straightforward approach to dealing with the external environment. These features improve KLEE's performance by over an order of magnitude and let it check a broad range of system-intensive programs "out of the box."

*Author names are in alphabetical order. Daniel Dunbar is the main author of the KLEE system.

Second, we show that KLEE's automatically-generated tests get high coverage on a diverse set of real, complicated, and environmentally-intensive programs. Our most in-depth evaluation applies KLEE to all 89 programs [1] in the latest stable version of GNU COREUTILS (version 6.10), which contains roughly 80,000 lines of library code and 61,000 lines in the actual utilities [2]. These programs interact extensively with their environment to provide a variety of functions, including managing the file system (e.g., `ls`, `dd`, `chmod`), displaying and configuring system properties (e.g., `logname`, `printenv`, `hostname`), controlling command invocation (e.g., `nohup`, `nice`, `env`), processing text files (e.g., `sort`, `od`, `patch`), and so on. They form the core user-level environment installed on many Unix systems. They are used daily by millions of people, bug fixes are handled promptly, and new releases are pushed regularly. Moreover, their extensive interaction with the environment stress-tests symbolic execution where it has historically been weakest.

Further, finding bugs in COREUTILS is hard. They are arguably the single most well-tested suite of open-source applications available (e.g., is there a program the reader has used more under Unix than "`ls`"?). In 1995, random testing of a subset of COREUTILS utilities found markedly fewer failures as compared to seven commercial Unix systems [35]. The last COREUTILS vulnerability reported on the SecurityFocus or US National Vulnerability databases was three years ago [5, 7].

In addition, we checked two other UNIX utility suites: BUSYBOX, a widely-used distribution for embedded systems [1], and the latest release for MINIX [4]. Finally, we checked the HISTAR operating system kernel as a contrast to application code [39].

Our experiments fall into three categories: (1) those where we do intensive runs to both find bugs and get high coverage (COREUTILS, HISTAR, and 75 BUSYBOX utilities), (2) those where we quickly run over many applications to find bugs (an additional 204 BUSYBOX utilities and 77 MINIX utilities), and (3) those where we crosscheck equivalent programs to find deeper correctness bugs (67 BUSYBOX utilities vs. the equivalent 67 in COREUTILS).

In total, we ran KLEE on more than 452 programs, containing over 430K total lines of code. To the best of our knowledge, this represents an order of magnitude more code and distinct programs than checked by prior symbolic test generation work. Our experiments show:

1 KLEE gets high coverage on a broad set of complex programs. Its automatically generated tests covered 84.5% of the total lines in COREUTILS and 90.5% in BUSYBOX (ignoring library code). On average these tests hit over 90% of the lines in each tool (median: over 94%), achieving perfect 100% coverage in 16 COREUTILS tools and 31 BUSYBOX tools.

2 KLEE can get significantly more code coverage than a concentrated, sustained manual effort. The roughly 89-hour run used to generate COREUTILS line coverage beat the developers' own test suite — built incrementally over fifteen years — by 16.8%!

3 With one exception, KLEE achieved these high-coverage results on unaltered applications. The sole exception, `sort` in COREUTILS, required a single edit to shrink a large buffer that caused problems for the constraint solver.

4 KLEE finds important errors in heavily-tested code. It found ten fatal errors in COREUTILS (including three that had escaped detection for 15 years), which account for more crashing bugs than were reported in 2006, 2007 and 2008 combined. It further found 24 bugs in BUSYBOX, 21 bugs in MINIX, and a security vulnerability in HISTAR– a total of 56 serious bugs.

5 The fact that KLEE test cases can be run on the raw version of the code (e.g., compiled with `gcc`) greatly simplifies debugging and error reporting. For example, all COREUTILS bugs were confirmed and fixed within two days and versions of the tests KLEE generated were included in the standard regression suite.

6 KLEE is not limited to low-level programming errors: when used to crosscheck purportedly identical BUSYBOX and GNU COREUTILS tools, it automatically found functional correctness errors and a myriad of inconsistencies.

7 KLEE can also be applied to non-application code. When applied to the core of the HISTAR kernel, it achieved an average line coverage of 76.4% (with disk) and 67.1% (without disk) and found a serious security bug.

The next section gives an overview of our approach. Section 3 describes KLEE, focusing on its key optimizations. Section 4 discusses how to model the environment. The heart of the paper is Section 5, which presents our experimental results. Finally, Section 6 describes related work and Section 7 concludes.

2 Overview

This section explains how KLEE works by walking the reader through the testing of MINIX's `tr` tool. Despite its small size — 169 lines, 83 of which are executable — it illustrates two problems common to the programs we check:

1 *Complexity.* The code aims to translate and delete characters from its input. It hides this intent well beneath non-obvious input parsing code, tricky boundary conditions, and hard-to-follow control flow. Figure 1 gives a representative snippet.

[1] We ignored utilities that are simply wrapper calls to others, such as `arch` ("`uname -m`") and `vdir` ("`ls -l -b`").

2 *Environmental Dependencies.* Most of the code is controlled by values derived from environmental input. Command line arguments determine what procedures execute, input values determine which way if-statements trigger, and the program depends on the ability to read from the file system. Since inputs can be invalid (or even malicious), the code must handle these cases gracefully. It is not trivial to test all important values and boundary cases.

The code illustrates two additional common features. First, it has bugs, which KLEE finds and generates test cases for. Second, KLEE quickly achieves good code coverage: in two minutes it generates 37 tests that cover all executable statements. [2]

KLEE has two goals: (1) hit every line of executable code in the program and (2) detect at each dangerous operation (e.g., dereference, assertion) if *any* input value exists that could cause an error. KLEE does so by running programs *symbolically*: unlike normal execution, where operations produce concrete values from their operands, here they generate constraints that exactly describe the set of values possible on a given path. When KLEE detects an error or when a path reaches an `exit` call, KLEE solves the current path's constraints (called its *path condition*) to produce a test case that will follow the same path when rerun on an unmodified version of the checked program (e.g, compiled with `gcc`).

KLEE is designed so that the paths followed by the unmodified program will always follow the same path KLEE took (i.e., there are no false positives). However, non-determinism in checked code and bugs in KLEE or its models have produced false positives in practice. The ability to rerun tests outside of KLEE, in conjunction with standard tools such as `gdb` and `gcov` is invaluable for diagnosing such errors and for validating our results.

We next show how to use KLEE, then give an overview of how it works.

2.1 Usage

A user can start checking many real programs with KLEE in seconds: KLEE typically requires no source modifications or manual work. Users first compile their code to bytecode using the publicly-available LLVM compiler [33] for GNU C. We compiled `tr` using:

```
llvm-gcc --emit-llvm -c tr.c -o tr.bc
```

Users then run KLEE on the generated bytecode, optionally stating the number, size, and type of symbolic inputs to test the code on. For `tr` we used the command:

```
klee --max-time 2 --sym-args 1 10 10
   --sym-files 2 2000 --max-fail 1 tr.bc
```

[2]The program has one line of dead code, an unreachable return statement, which, reassuringly, KLEE cannot run.

```
1 : void expand(char *arg, unsigned char *buffer) {          8
2 :   int i, ac;                                             9
3 :   while (*arg) {                                         10*
4 :     if (*arg == '\\') {                                  11*
5 :       arg++;
6 :       i = ac = 0;
7 :       if (*arg >= '0' && *arg <= '7') {
8 :         do {
9 :           ac = (ac << 3) + *arg++ - '0';
10:           i++;
11:         } while (i<4 && *arg>='0' && *arg<='7');
12:         *buffer++ = ac;
13:       } else if (*arg != '\0')
14:         *buffer++ = *arg++;
15:     } else if (*arg == '[') {                            12*
16:       arg++;                                             13
17:       i = *arg++;                                        14
18:       if (*arg++ != '-') {                               15!
19:         *buffer++ = '[';
20:         arg -= 2;
21:         continue;
22:       }
23:       ac = *arg++;
24:       while (i <= ac) *buffer++ = i++;
25:       arg++;       /* Skip ']' */
26:     } else
27:       *buffer++ = *arg++;
28:   }
29: }
30: ...
31: int main(int argc, char* argv[]) {                        1
32:   int index = 1;                                         2
33:   if (argc > 1 && argv[index][0] == '-') {               3*
34:     ...                                                  4
35:   }                                                      5
36:   ...                                                    6
37:   expand(argv[index++], index);                          7
38:   ...
39: }
```

Figure 1: Code snippet from MINIX's `tr`, representative of the programs checked in this paper: tricky, non-obvious, difficult to verify by inspection or testing. The order of the statements on the path to the error at line 18 are numbered on the right hand side.

The first option, `--max-time`, tells KLEE to check `tr.bc` for at most two minutes. The rest describe the symbolic inputs. The option `--sym-args 1 10 10` says to use zero to three command line arguments, the first 1 character long, the others 10 characters long. [3] The option `--sym-files 2 2000` says to use standard input and one file, each holding 2000 bytes of symbolic data. The option `--max-fail 1` says to fail at most one system call along each program path (see § 4.2).

2.2 Symbolic execution with KLEE

When KLEE runs on `tr`, it finds a buffer overflow error at line 18 in Figure 1 and then produces a concrete test

[3]Since strings in C are zero terminated, this essentially generates arguments of *up to* that size.

case (`tr [ "" ""`) that hits it. Assuming the options of the previous subsection, KLEE runs `tr` as follows:

1 KLEE constructs symbolic command line string arguments whose contents have no constraints other than zero-termination. It then constrains the number of arguments to be between 0 and 3, and their sizes to be 1, 10 and 10 respectively. It then calls `main` with these initial path constraints.
2 When KLEE hits the branch `argc > 1` at line 33, it uses its constraint solver STP [23] to see which directions can execute given the current path condition. For this branch, both directions are possible; KLEE forks execution and follows both paths, adding the constraint $\texttt{argc} > 1$ on the false path and $\texttt{argc} \leq 1$ on the true path.
3 Given more than one active path, KLEE must pick which one to execute first. We describe its algorithm in Section 3.4. For now assume it follows the path that reaches the bug. As it does so, KLEE adds further constraints to the contents of `arg`, and forks for a total of five times (lines denoted with a "`*`"): twice on line 33, and then on lines 3, 4, and 15 in `expand`.
4 At each dangerous operation (e.g., pointer dereference), KLEE checks if any possible value allowed by the current path condition would cause an error. On the annotated path, KLEE detects no errors before line 18. At that point, however, it determines that input values exist that allow the read of `arg` to go out of bounds: after taking the true branch at line 15, the code increments `arg` twice without checking if the string has ended. If it has, this increment skips the terminating `'\0'` and points to invalid memory.
5 KLEE generates concrete values for `argc` and `argv` (i.e., `tr [ "" ""`) that when rerun on a raw version of `tr` will hit this bug. It then continues following the current path, adding the constraint that the error does not occur (in order to find other errors).

3 The KLEE Architecture

KLEE is a complete redesign of our previous system EXE [16]. At a high level, KLEE functions as a hybrid between an operating system for symbolic processes and an interpreter. Each symbolic process has a register file, stack, heap, program counter, and path condition. To avoid confusion with a Unix process, we refer to KLEE's representation of a symbolic process as a *state*. Programs are compiled to the LLVM [33] assembly language, a RISC-like virtual instruction set. KLEE directly interprets this instruction set, and maps instructions to constraints without approximation (i.e. bit-level accuracy). [4]

[4] KLEE does not currently support: symbolic floating point, `longjmp`, threads, and assembly code. Additionally, memory objects are required to have concrete sizes.

3.1 Basic architecture

At any one time, KLEE may be executing a large number of states. The core of KLEE is an interpreter loop which selects a state to run and then symbolically executes a single instruction in the context of that state. This loop continues until there are no states remaining, or a user-defined timeout is reached.

Unlike a normal process, storage locations for a state — registers, stack and heap objects — refer to expressions (trees) instead of raw data values. The leaves of an expression are symbolic variables or constants, and the interior nodes come from LLVM assembly language operations (e.g., arithmetic operations, bitwise manipulation, comparisons, and memory accesses). Storage locations which hold a constant expression are said to be *concrete*.

Symbolic execution of the majority of instructions is straightforward. For example, to symbolically execute an LLVM add instruction:

```
%dst = add i32 %src0, %src1
```

KLEE retrieves the addends from the `%src0` and `%src1` registers and writes a new expression `Add(%src0, %src1)` to the `%dst` register. For efficiency, the code that builds expressions checks if all given operands are concrete (i.e., constants) and, if so, performs the operation natively, returning a constant expression.

Conditional branches take a boolean expression (branch condition) and alter the instruction pointer of the state based on whether the condition is true or false. KLEE queries the constraint solver to determine if the branch condition is either provably true or provably false along the current path; if so, the instruction pointer is updated to the appropriate location. Otherwise, both branches are possible: KLEE clones the state so that it can explore both paths, updating the instruction pointer and path condition on each path appropriately.

Potentially dangerous operations implicitly generate branches that check if any input value exists that could cause an error. For example, a division instruction generates a branch that checks for a zero divisor. Such branches work identically to normal branches. Thus, even when the check succeeds (i.e., an error is detected), execution continues on the false path, which adds the negation of the check as a constraint (e.g., making the divisor not zero). If an error is detected, KLEE generates a test case to trigger the error and terminates the state.

As with other dangerous operations, load and store instructions generate checks: in this case to check that the address is in-bounds of a valid memory object. However, load and store operations present an additional complication. The most straightforward representation of the memory used by checked code would be a flat byte array. In this case, loads and stores would simply map to

array read and write expressions respectively. Unfortunately, our constraint solver STP would almost never be able to solve the resultant constraints (and neither would the other constraint solvers we know of). Thus, as in EXE, KLEE maps every memory object in the checked code to a distinct STP array (in a sense, mapping a flat address space to a segmented one). This representation dramatically improves performance since it lets STP ignore all arrays not referenced by a given expression.

Many operations (such as bound checks or object-level copy-on-write) require object-specific information. If a pointer can refer to many objects, these operations become difficult to perform. For simplicity, KLEE sidesteps this problem as follows. When a dereferenced pointer `p` can refer to N objects, KLEE clones the current state N times. In each state it constrains `p` to be within bounds of its respective object and then performs the appropriate read or write operation. Although this method can be expensive for pointers with large points-to sets, most programs we have tested only use symbolic pointers that refer to a single object, and KLEE is well-optimized for this case.

3.2 Compact state representation

The number of states grows quite quickly in practice: often even small programs generate tens or even hundreds of thousands of concurrent states during the first few minutes of interpretation. When we ran COREUTILS with a 1GB memory cap, the maximum number of concurrent states recorded was 95,982 (for `hostid`), and the average of this maximum for each tool was 51,385. This explosion makes state size critical.

Since KLEE tracks all memory objects, it can implement copy-on-write at the object level (rather than page granularity), dramatically reducing per-state memory requirements. By implementing the heap as an immutable map, portions of the heap structure itself can also be shared amongst multiple states (similar to sharing portions of page tables across `fork()`). Additionally, this heap structure can be cloned in constant time, which is important given the frequency of this operation.

This approach is in marked contrast to EXE, which used one native OS process per state. Internalizing the state representation dramatically increased the number of states which can be concurrently explored, both by decreasing the per-state cost and allowing states to share memory at the object (rather than page) level. Additionally, this greatly simplified the implementation of caches and search heuristics which operate across all states.

3.3 Query optimization

Almost always, the cost of constraint solving dominates everything else — unsurprising, given that KLEE generates complicated queries for an NP-complete logic. Thus, we spent a lot of effort on tricks to simplify expressions and ideally eliminate queries (no query is the fastest query) before they reach STP. Simplified queries make solving faster, reduce memory consumption, and increase the query cache's hit rate (see below). The main query optimizations are:

Expression Rewriting. The most basic optimizations mirror those in a compiler: e.g., simple arithmetic simplifications (`x + 0 = 0`), strength reduction (`x * 2^n = x << n`), linear simplification (`2*x - x = x`).

Constraint Set Simplification. Symbolic execution typically involves the addition of a large number of constraints to the path condition. The natural structure of programs means that constraints on same variables tend to become more specific. For example, commonly an inexact constraint such as $x < 10$ gets added, followed some time later by the constraint $x = 5$. KLEE actively simplifies the constraint set by rewriting previous constraints when new equality constraints are added to the constraint set. In this example, substituting the value for x into the first constraint simplifies it to `true`, which KLEE eliminates.

Implied Value Concretization. When a constraint such as $x + 1 = 10$ is added to the path condition, then the value of `x` has effectively become concrete along that path. KLEE determines this fact (in this case that $x = 9$) and writes the concrete value back to memory. This ensures that subsequent accesses of that memory location can return a cheap constant expression.

Constraint Independence. Many constraints do not overlap in terms of the memory they reference. Constraint independence (taken from EXE) divides constraint sets into disjoint independent subsets based on the symbolic variables they reference. By explicitly tracking these subsets, KLEE can frequently eliminate irrelevant constraints prior to sending a query to the constraint solver. For example, given the constraint set $\{i < j, j < 20, k > 0\}$, a query of whether $i = 20$ just requires the first two constraints.

Counter-example Cache. Redundant queries are frequent, and a simple cache is effective at eliminating a large number of them. However, it is possible to build a more sophisticated cache due to the particular structure of constraint sets. The counter-example cache maps sets of constraints to counter-examples (i.e., variable assignments), along with a special sentinel used when a set of constraints has no solution. This mapping is stored in a custom data structure — derived from the UBTree structure of Hoffmann and Hoehler [28] — which allows efficient searching for cache entries for both subsets and supersets of a constraint set. By storing the cache in this fashion, the counter-example cache gains three additional ways to eliminate queries. In the example below, we assume that the counter-example cache

Optimizations	Queries	Time (s)	STP Time (s)
None	13717	300	281
Independence	13717	166	148
Cex. Cache	8174	177	156
All	699	20	10

Table 1: Performance comparison of KLEE's solver optimizations on COREUTILS. Each tool is run for 5 minutes without optimization, and rerun on the same workload with the given optimizations. The results are averaged across all applications.

currently has entries for $\{i < 10, i = 10\}$ (no solution) and $\{i < 10, j = 8\}$ (satisfiable, with variable assignments $i \rightarrow 5, j \rightarrow 8$).

1 When a subset of a constraint set has no solution, then neither does the original constraint set. Adding constraints to an unsatisfiable constraint set cannot make it satisfiable. For example, given the cache above, $\{i < 10, i = 10, j = 12\}$ is quickly determined to be unsatisfiable.

2 When a superset of a constraint set has a solution, that solution also satisfies the original constraint set. Dropping constraints from a constraint set does not invalidate a solution to that set. The assignment $i \rightarrow 5, j \rightarrow 8$, for example, satisfies either $i < 10$ or $j = 8$ individually.

3 When a subset of a constraint set has a solution, it is likely that this is also a solution for the original set. This is because the extra constraints often do not invalidate the solution to the subset. Because checking a potential solution is cheap, KLEE tries substituting in all solutions for subsets of the constraint set and returns a satisfying solution, if found. For example, the constraint set $\{i < 10, j = 8, i \neq 3\}$ can still be satisfied by $i \rightarrow 5, j \rightarrow 8$.

To demonstrate the effectiveness of these optimizations, we performed an experiment where COREUTILS applications were run for 5 minutes with both of these optimizations turned off. We then deterministically reran the exact same workload with constraint independence and the counter-example cache enabled separately and together for the same number of instructions. This experiment was done on a large sample of COREUTILS utilities. The results in Table 1 show the averaged results.

As expected, the independence optimization by itself does not eliminate any queries, but the simplifications it performs reduce the overall running time by almost half (45%). The counter-example cache reduces both the running time and the number of STP queries by 40%. However, the real win comes when both optimizations are enabled; in this case the hit rate for the counter-example cache greatly increases due to the queries first being simplified via independence. For the sample runs, the average number of STP queries are reduced to 5% of the original number and the average runtime decreases by more than an order of magnitude.

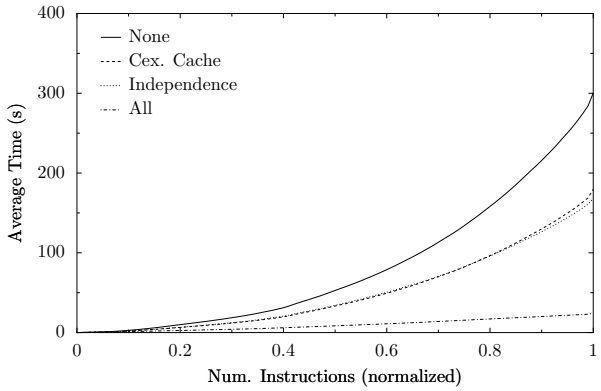


Figure 2: The effect of KLEE's solver optimizations over time, showing they become more effective over time, as the caches fill and queries become more complicated. The number of executed instructions is normalized so that data can be aggregated across all applications.

It is also worth noting the degree to which STP time (time spent solving queries) dominates runtime. For the original runs, STP accounts for 92% of overall execution time on average (the combined optimizations reduce this by almost 300%). With both optimizations enabled this percentage drops to 41%. Finally, Figure 2 shows the efficacy of KLEE's optimizations increases with time — as the counter-example cache is filled and query sizes increase, the speed-up from the optimizations also increases.

3.4 State scheduling

KLEE selects the state to run at each instruction by interleaving the following two search heuristics.

Random Path Selection maintains a binary tree recording the program path followed for all active states, i.e. the leaves of the tree are the current states and the internal nodes are places where execution forked. States are selected by traversing this tree from the root and randomly selecting the path to follow at branch points. Therefore, when a branch point is reached, the set of states in each subtree has equal probability of being selected, regardless of the size of their subtrees. This strategy has two important properties. First, it favors states high in the branch tree. These states have less constraints on their symbolic inputs and so have greater freedom to reach uncovered code. Second, and most importantly, this strategy avoids starvation when some part of the program is rapidly creating new states ("fork bombing") as it happens when a tight loop contains a symbolic condition. Note that the simply selecting a state at random has neither property.

Coverage-Optimized Search tries to select states likely to cover new code in the immediate future. It uses heuristics to compute a weight for each state and then randomly selects a state according to these weights. Currently these heuristics take into account the minimum distance to an uncovered instruction, the call stack of the state, and whether the state recently covered new code.

KLEE uses each strategy in a round robin fashion. While this can increase the time for a particularly effective strategy to achieve high coverage, it protects against cases where an individual strategy gets stuck. Furthermore, since strategies pick from the same state pool, interleaving them can improve overall effectiveness.

The time to execute an individual instruction can vary widely between simple instructions (e.g., addition) and instructions which may use the constraint solver or fork execution (branches, memory accesses). KLEE ensures that a state which frequently executes expensive instructions will not dominate execution time by running each state for a "time slice" defined by both a maximum number of instructions and a maximum amount of time.

4 Environment Modeling

When code reads values from its environment — command-line arguments, environment variables, file data and metadata, network packets, etc — we conceptually want to return all values that the read could legally produce, rather than just a single concrete value. When it writes to its environment, the effects of these alterations should be reflected in subsequent reads. The combination of these features allows the checked program to explore all potential actions and still have no false positives.

Mechanically, we handle the environment by redirecting calls that access it to *models* that understand the semantics of the desired action well enough to generate the required constraints. Crucially, these models are written in normal C code which the user can readily customize, extend, or even replace without having to understand the internals of KLEE. We have about 2,500 lines of code to define simple models for roughly 40 system calls (e.g., `open`, `read`, `write`, `stat`, `lseek`, `ftruncate`, `ioctl`).

4.1 Example: modeling the file system

For each file system operation we check if the action is for an actual concrete file on disk or a symbolic file. For concrete files, we simply invoke the corresponding system call in the running operating system. For symbolic files we emulate the operation's effect on a simple symbolic file system, private to each state.

Figure 3 gives a rough sketch of the model for `read()`, eliding details for dealing with linking, reads on standard input, and failures. The code maintains a set of file descriptors, created at file `open()`, and records

```
1 : ssize_t read(int fd, void *buf, size_t count) {
2 :   if (is_invalid(fd)) {
3 :     errno = EBADF;
4 :     return -1;
5 :   }
6 :   struct klee_fd *f = &fds[fd];
7 :   if (is_concrete_file(f)) {
8 :     int r = pread(f->real_fd, buf, count, f->off);
9 :     if (r != -1)
10:       f->off += r;
11:     return r;
12:   } else {
13:     /* sym files are fixed size: don't read beyond the end. */
14:     if (f->off >= f->size)
15:       return 0;
16:     count = min(count, f->size - f->off);
17:     memcpy(buf, f->file_data + f->off, count);
18:     f->off += count;
19:     return count;
20:   }
21: }
```

Figure 3: Sketch of KLEE's model for `read()`.

for each whether the associated file is symbolic or concrete. If `fd` refers to a concrete file, we use the operating system to read its contents by calling `pread()` (lines 7-11). We use `pread` to multiplex access from KLEE's many states onto the one actual underlying file descriptor. [5] If `fd` refers to a symbolic file, `read()` copies from the underlying symbolic buffer holding the file contents into the user supplied buffer (lines 13-19). This ensures that multiple `read()` calls that access the same file use consistent symbolic values.

Our symbolic file system is crude, containing only a single directory with N symbolic files in it. KLEE users specify both the number N and the size of these files. This symbolic file system coexists with the real file system, so applications can use both symbolic and concrete files. When the program calls `open` with a concrete name, we (attempt to) open the actual file. Thus, the call:

```
int fd = open("/etc/fstab", O_RDNLY);
```

sets `fd` to point to the actual configuration file `/etc/fstab`.

On the other hand, calling `open()` with an unconstrained symbolic name matches each of the N symbolic files in turn, and will also fail once. For example, given $N = 1$, calling `open()` with a symbolic command-line argument `argv[1]`:

```
int fd = open(argv[1], O_RDNLY);
```

will result in two paths: one in which `fd` points to the single symbolic file in the environment, and one in which `fd` is set to `-1` indicating an error.

[5] Since KLEE's states execute within a single Unix process (the one used to run KLEE), then unless we duplicated file descriptors for each (which seemed expensive), a `read` by one would affect all the others.

Unsurprisingly, the choice of what interface to model has a big impact on model complexity. Rather than having our models at the system call level, we could have instead built them at the C standard library level (`fopen`, `fread`, etc.). Doing so has the potential performance advantage that, for concrete code, we could run these operations natively. The major downside, however, is that the standard library contains a huge number of functions — writing models for each would be tedious and error-prone. By only modeling the much simpler, low-level system call API, we can get the richer functionality by just compiling one of the many implementations of the C standard library (we use uClibc [6]) and let it worry about correctness. As a side-effect, we simultaneously check the library for errors as well.

4.2 Failing system calls

The real environment can fail in unexpected ways (e.g., `write()` fails because of a full disk). Such failures can often lead to unexpected and hard to diagnose bugs. Even when applications do try to handle them, this code is rarely fully exercised by the regression suite. To help catch such errors, KLEE will optionally simulate environmental failures by failing system calls in a controlled manner (similar to [38]). We made this mode optional since not all applications care about failures — a simple application may ignore disk crashes, while a mail server expends a lot of code to handle them.

4.3 Rerunning test cases

KLEE-generated test cases are rerun on the unmodified native binaries by supplying them to a replay driver we provide. The individual test cases describe an instance of the symbolic environment. The driver uses this description to create actual operating system objects (files, pipes, ttys, directories, links, etc.) containing the concrete values used in the test case. It then executes the unmodified program using the concrete command-line arguments from the test case. Our biggest challenge was making system calls fail outside of KLEE — we built a simple utility that uses the `ptrace` debugging interface to skip the system calls that were supposed to fail and instead return an error.

5 Evaluation

This section describes our in-depth coverage experiments for COREUTILS (§ 5.2) and BUSYBOX (§ 5.3) as well as errors found during quick bug-finding runs (§ 5.4). We use KLEE to find deep correctness errors by crosschecking purportedly equivalent tool implementations (§ 5.5) and close with results for HISTAR (§5.6).

5.1 Coverage methodology

We use line coverage as a conservative measure of KLEE-produced test case effectiveness. We chose executable line coverage as reported by `gcov`, because it is widely-understood and uncontroversial. Of course, it grossly underestimates KLEE's thoroughness, since it ignores the fact that KLEE explores many different unique paths with all possible values. We expect a path-based metric would show even more dramatic wins.

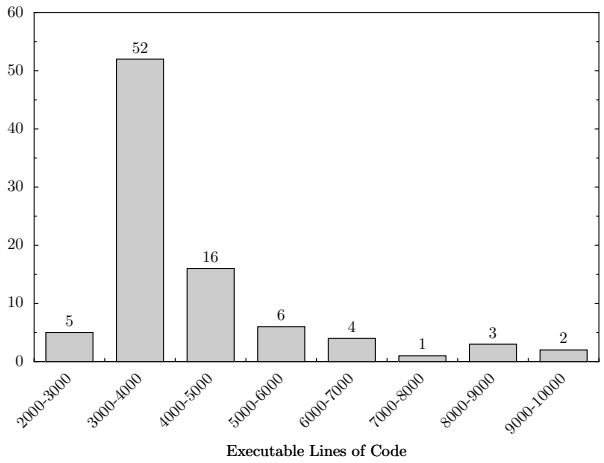


Figure 4: Histogram showing the number of COREUTILS tools that have a given number of executable lines of code (ELOC).

We measure coverage by running KLEE-generated test cases on a stand-alone version of each utility and using `gcov` to measure coverage. Running tests independently of KLEE eliminates the effect of bugs in KLEE and verifies that the produced test case runs the code it claims.

Note, our coverage results only consider code in the tool itself. They do not count library code since doing so makes the results harder to interpret:

1. It double-counts many lines, since often the same library function is called by many applications.
2. It unfairly under-counts coverage. Often, the bulk of a library function called by an application is effectively dead code since the library code is general but call sites are not. For example, `printf` is exceptionally complex, but the call `printf("hello")` can only hit a small a fraction (missing the code to print integers, floating point, formatting, etc.).

However, we do include library code when measuring the raw size of the application: KLEE must successfully handle this library code (and gets no credit for doing so) in order to exercise the code in the tool itself. We measure size in terms of executable lines of code (ELOC) by counting the total number of executable lines in the final executable after global optimization, which eliminates uncalled functions and other dead code. This measure is usually a factor of three smaller than a simple line count (using `wc -l`).

In our experiments KLEE minimizes the test cases it

Coverage (w/o lib)	COREUTILS		BUSYBOX	
	KLEE tests	Devel. tests	KLEE tests	Devel. tests
100%	16	1	31	4
90-100%	40	6	24	3
80-90%	21	20	10	15
70-80%	7	23	5	6
60-70%	5	15	2	7
50-60%	-	10	-	4
40-50%	-	6	-	-
30-40%	-	3	-	2
20-30%	-	1	-	1
10-20%	-	3	-	-
0-10%	-	1	-	30
Overall cov.	84.5%	67.7%	90.5%	44.8%
Med cov/App	94.7%	72.5%	97.5%	58.9%
Ave cov/App	90.9%	68.4%	93.5%	43.7%

Table 2: Number of COREUTILS tools which achieve line coverage in the given ranges for KLEE and developers' tests (library code not included). The last rows shows the aggregate coverage achieved by each method and the average and median coverage per application.

generates by only emitting tests cases for paths that hit a new statement or branch in the main utility code. A user that wants high library coverage can change this setting.

5.2 GNU COREUTILS

We now give KLEE coverage results for all 89 GNU COREUTILS utilities.

Figure 4 breaks down the tools by executable lines of code (ELOC), including library code the tool calls. While relatively small, the tools are not toys — the smallest five have between 2K and 3K ELOC, over half (52) have between 3K and 4K, and ten have over 6K.

Previous work, ours included, has evaluated constraint-based execution on a small number of hand-selected benchmarks. Reporting results for the entire COREUTILS suite, the worst along with the best, prevents us from hand-picking results or unintentionally cheating through the use of fragile optimizations.

Almost all tools were tested using the same command (command arguments explained in § 2.1):

```
./run <tool-name>  --max-time 60
                   --sym-args 10 2 2
                   --sym-files 2 8
                   [--max-fail 1]
```

As specified by the `--max-time` option, we ran each tool for about 60 minutes (some finished before this limit, a few up to three minutes after). For eight tools where the coverage results of these values were unsatisfactory, we consulted the `man` page and increased the number and size of arguments and files. We found this easy to do, so presumably a tool implementer or user would as well. After these runs completed, we improved them by failing system calls (see § 4.2).

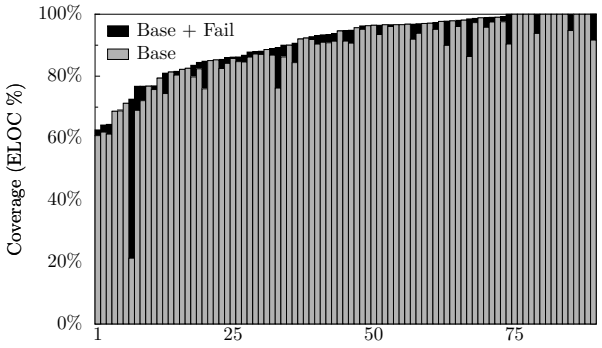


Figure 5: Line coverage for each application with and without failing system calls.

5.2.1 Line coverage results

The first two columns in Table 2 give aggregate line coverage results. On average our tests cover 90.9% of the lines in each tool (median: 94.7%), with an overall (aggregate) coverage across all tools of 84.5%. We get 100% line coverage on 16 tools, over 90% on 56 tools, and over 80% on 77 tools (86.5% of all tools). The minimum coverage achieved on any tool is 62.6%.

We believe such high coverage on a broad swath of applications "out of the box" convincingly shows the power of the approach, especially since it is across the entire tool suite rather than focusing on a few particular applications.

Importantly, KLEE generates high coverage with few test cases: for our non-failing runs, it needs a total of 3,321 tests, with a per-tool average of 37 (median: 33). The maximum number needed was 129 (for the "`[`" tool) and six needed 5. As a crude measure of path complexity, we counted the number of static branches run by each test case using `gcov`[6] (i.e., an executed branch counts once no matter how many times the branch ran dynamically). The average path length was 76 (median: 53), the maximum was 512 and (to pick a random number) 160 were at least 250 branches long.

Figure 5 shows the coverage KLEE achieved on each tool, with and without failing system call invocations. Hitting system call failure paths is useful for getting the last few lines of high-coverage tools, rather than significantly improving the overall results (which it improves from 79.9% to 84.5%). The one exception is `pwd` which requires system call failures to go from a dismal 21.2% to 72.6%. The second best improvement for a single tool is a more modest 13.1% extra coverage on the `df` tool.

[6] In `gcov` terminology, a branch is a possible branch direction, i.e. a simple if statement has two branches.

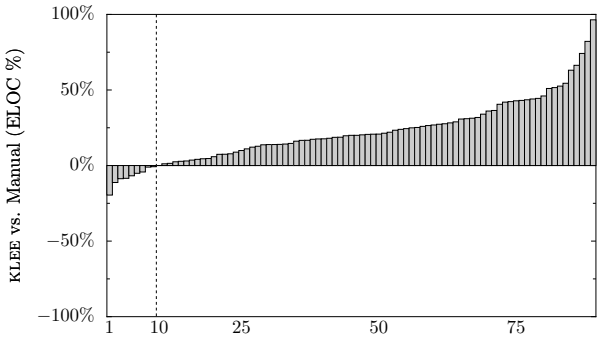


Figure 6: Relative coverage difference between KLEE and the COREUTILS manual test suite, computed by subtracting the executable lines of code covered by manual tests (L_{man}) from KLEE tests (L_{klee}) and dividing by the total possible: $(L_{klee} - L_{man})/L_{total}$. Higher bars are better for KLEE, which beats manual testing on all but 9 applications, often significantly.

```
paste -d\\ abcdefghijklmnopqrstuvwxyz
pr -e t2.txt
tac -r t3.txt t3.txt
mkdir -Z a b
mkfifo -Z a b
mknod -Z a b p
md5sum -c t1.txt
ptx -F\\ abcdefghijklmnopqrstuvwxyz
ptx x t4.txt
seq -f %0 1
```

t1.txt: `"\t \tMD5("`
t2.txt: `"\b\b\b\b\b\b\b\t"`
t3.txt: `"\n"`
t4.txt: `"a"`

Figure 7: KLEE-generated command lines and inputs (modified for readability) that cause program crashes in COREUTILS version 6.10 when run on Fedora Core 7 with SELinux on a Pentium machine.

5.2.2 Comparison against developer test suites

Each utility in COREUTILS comes with an extensive manually-written test suite extended each time a new bug fix or extra feature is added. [7] As Table 2 shows, KLEE beats developer tests handily on all aggregate measures: overall total line coverage (84.5% versus 67.7%), average coverage per tool (90.9% versus 68.4%) and median coverage per tool (94.7% versus 72.5%). At a more detailed level, KLEE gets 100% coverage on 16 tools and over 90% coverage on 56 while the developer tests get 100% on a single utility (`true`) and reach over 90% on only 7. Finally, the developers tests get below 60% coverage on 24 tools while KLEE always achieves over 60%. In total, an 89 hour run of KLEE (about one hour per application) exceeds the coverage of a test suite built over a period of fifteen years by 16.8%!

Figure 6 gives a relative view of KLEE versus developer tests by subtracting the lines hit by manual testing from those hit by KLEE and dividing this by the total possible. A bar above zero indicates that KLEE beat the manual test (and by how much); a bar below shows the opposite. KLEE beats manual testing, often significantly, on the vast majority of the applications.

To guard against hidden bias in line coverage, we also compared the taken branch coverage (as reported by `gcov`) of the manual and KLEE test suites. While the absolute coverage for both test suites decreases, KLEE's relative improvement over the developers' tests remains: KLEE achieves 76.9% overall branch coverage, while the developers' tests get only 56.5%.

Finally, it is important to note that although KLEE's runs significantly beat the developers' tests in terms of coverage, KLEE only checks for low-level errors and violations of user-level asserts. In contrast, developer tests typically validate that the application output matches the expected one. We partially address this limitation by validating the output of these utilities against the output produces by a different implementation (see § 5.5).

5.2.3 Bugs found

KLEE found ten unique bugs in COREUTILS (usually memory error crashes). Figure 7 gives the command lines used to trigger them. The first three errors existed since at least 1992, so should theoretically crash any COREUTILS distribution up to 6.10. The others are more recent, and do not crash older COREUTILS distributions. While one bug (in `seq`) had been fixed in the developers' unreleased version, the other bugs were confirmed and fixed within two days of our report. In addition, versions of the KLEE-generated test cases for the new bugs were added to the official COREUTILS test suite.

As an illustrative example, we discuss the bug in `pr` (used to paginate files before printing) hit by the invocation "`pr -e t2.txt`" in Figure 7. The code containing the bug is shown in Figure 8. On the path that hits the bug, both `chars_per_input_tab` and `chars_per_c` equal tab width (let's call it T). Line 2665 computes `width` $= (T -$ `input_position` $\bmod T)$ using the macro on line 602. The root cause of the bug is the incorrect assumption that $0 \leq x \mod y < y$, which only holds for positive integers. When `input_position` is positive, `width` will be less than T since $0 \leq$ `input_position` $\bmod T < T$. However, in the presence of backspaces, `input_position` can become neg-

[7] We ran the test suite using the commands: `env RUN_EXPENSIVE_TESTS=YES RUN_VERY_EXPENSIVE_TESTS=YES make check` and `make check-root` (as root). A small number of tests (14 out of 393) which require special configuration were not run; from manual inspection we do not expect these to have a significant impact on our results.

```
 602: #define TAB_WIDTH(c_, h_) ((c_) - ((h_) % (c_)))
...
1322: clump_buff = xmalloc(MAX(8,chars_per_input_tab));
...     // (set s to clump_buff)
2665: width = TAB_WIDTH(chars_per_c, input_position);
2666:
2667: if (untabify_input)
2668: {
2669:   for (i = width; i; --i)
2670:     *s++ = ' ';
2671:   chars = width;
2672: }
```

Figure 8: Code snippet from `pr` where a memory overflow of `clump_buff` via pointer `s` is possible if `chars_per_input_tab == chars_per_c` and `input_position < 0`.

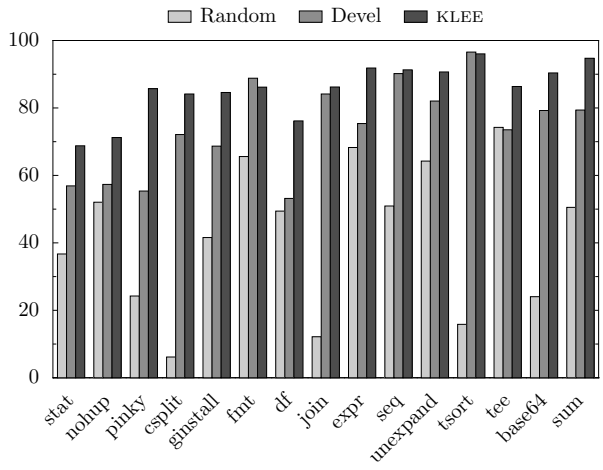


Figure 9: Coverage of random vs. manual vs. KLEE testing for 15 randomly-chosen COREUTILS utilities. Manual testing beats random on average, while KLEE beats both by a significant margin.

ative, so ($-T <$ `input_position` mod $T < T$). Consequently, `width` can be as large as $2T - 1$.

The bug arises when the code allocates a buffer `clump_buff` of size T (line 1322) and then writes `width` characters into this buffer (lines 2669–2670) via the pointer `s` (initially set to `clump_buff`). Because `width` can be as large as $2T - 1$, a memory overflow is possible.

This is a prime example of the power of symbolic execution in finding complex errors in code which is hard to reason about manually — this bug has existed in `pr` since at least 1992, when COREUTILS was first added to a CVS repository.

5.2.4 Comparison with random tests

In our opinion, the COREUTILS manual tests are unusually comprehensive. However, we compare to random testing both to guard against deficiencies, and to get a feel for how constraint-based reasoning compares to blind random guessing. We tried to make the comparison apples-to-apples by building a tool that takes the same command line as KLEE, and generates random values for the specified type, number, and size range of inputs. It then runs the checked program on these values using the same replay infrastructure as KLEE. For time reasons, we randomly chose 15 benchmarks (shown in Figure 9) and ran them for 65 minutes (to always exceed the time given to KLEE) with the same command lines used when run with KLEE.

Figure 9 shows the coverage for these programs achieved by random, manual, and KLEE tests. Unsurprisingly, given the complexity of COREUTILS programs and the concerted effort of the COREUTILS maintainers, the manual tests get significantly more coverage than random. KLEE handily beats both.

Because `gcov` introduces some overhead, we also performed a second experiment in which we ran each tool natively without `gcov` for 65 minutes (using the same random seed as the first run), recorded the number of test cases generated, and then reran using `gcov` for that number. This run completely eliminates the `gcov` overhead, and overall it generates 44% more tests than during the initial run.

However, these 44% extra tests increase the average coverage per tool by only 1%, with 11 out of 15 utilities not seeing any improvement — showing that random gets stuck for most applications. We have seen this pattern repeatedly in previous work: random quickly gets the cases it can, and then revisits them over and over. Intuitively, satisfying even a single 32-bit equality requires correctly guessing one value out of four billion. Correctly getting a sequence of such conditionals is hopeless. Utilities such as `csplit` (the worst performer), illustrate this dynamic well: their input has structure, and the difficulty of blindly guessing values that satisfy its rules causes most inputs to be rejected.

One unexpected result was that for 11 of these 15 programs, KLEE explores paths to termination (i.e., the checked code calls `exit()`) only a few times slower than random does! KLEE explored paths to termination in roughly the same time for three programs and, in fact, was actually faster for three others (`seq`, `tee`, and `nohup`). We were surprised by these numbers, because we had assumed a constraint-based tool would run orders of magnitude more slowly than raw testing on a per-path basis, but would have the advantage of exploring more unique paths over time (with all values) because it did not get stuck. While the overhead on four programs matched this expectation (where constraint solver overhead made paths ran 7x to 220x more slowly than native execution), the performance tradeoff for the oth-

ers is more nuanced. Assume we have a branch deep in the program. Covering both true and false directions using traditional testing requires running the program from start to finish twice: once for the true path and again for the false. In contrast, while KLEE runs each instruction more slowly than native execution, it only needs to run the instruction path before the branch once, since it forks execution at the branch point (a fast operation given its object-level copy-on-write implementation). As path length grows, this ability to avoid redundantly rerunning path prefixes gets increasingly important.

With that said, the reader should view the per-path costs of random and KLEE as very crude estimates. First, the KLEE infrastructure random uses to run tests adds about 13ms of per-test overhead, as compared to around 1ms for simply invoking a program from a script. This code runs each test case in a sandbox directory, makes a clean environment, and creates various system objects with random contents (e.g., files, pipes, tty's). It then runs the tested program with a watchdog to terminate infinite loops. While a dedicated testing tool must do roughly similar actions, presumably it could shave some milliseconds. However, this fixed cost matters only for short program runs, such as when the code exits with an error. In cases where random can actually make progress and explore deeper program paths, the inefficiency of rerunning path prefixes starts to dominate. Further, we conservatively compute the path completion rate for KLEE: when its time expires, roughly 30% of the states it has created are still alive, and we give it no credit for the work it did on them.

5.3 BUSYBOX utilities

BUSYBOX is a widely-used implementation of standard UNIX utilities for embedded systems that aims for small executable sizes [1]. Where there is overlap, it aims to replicate COREUTILS functionality, although often providing fewer features. We ran our experiments on a bug-patched version of BUSYBOX 1.10.2. We ran the 75 utilities [8] in the BUSYBOX "`coreutils`" subdirectory (14K lines of code, with another 16K of library code), using the same command lines as when checking COREUTILS, except we did not fail system calls.

As Table 2 shows, KLEE does even better than on COREUTILS: over 90.5% total line coverage, on average covering 93.5% per tool with a median of 97.5%. It got 100% coverage on 31 and over 90% on 55 utilities.

BUSYBOX has a less comprehensive manual test suite than COREUTILS (in fact, many applications don't seem to have any tests). Thus, KLEE beats the developers tests by roughly a factor of two: 90.5% total line coverage versus only 44.8% for the developers' suite. The developers do better on only one benchmark, `cp`.

[8] We are actually measuring coverage on 72 files because several utilities are implemented in the same file.

`date -I`
`ls --co`
`chown a.a -`
`kill -l a`
`setuidgid a ""`
`printf "% *" B`
`od t1.txt`
`od t2.txt`
`printf %`
`printf %Lo`
`tr [`
`tr [=`
`tr [a-z`

t1.txt: `a`
t2.txt: `A`
t3.txt: `\t\n`

`cut -f t3.txt`
`install --m`
`nmeter -`
`envdir`
`setuidgid`
`envuidgid`
`envdir -`
`arp -Ainet`
`tar tf_ /`
`top d`
`setarch "" ""`
`<full-path>/linux32`
`<full-path>/linux64`
`hexdump -e ""`
`ping6 -`

Figure 10: KLEE-generated command lines and inputs (modified for readability) that cause program crashes in BUSYBOX. When multiple applications crash because of the same shared (buggy) piece of code, we group them by shading.

5.4 Bug-finding: MINIX + all BUSYBOX tools

To demonstrate KLEE's applicability to bug finding, we used KLEE to check all 279 BUSYBOX tools and 84 MINIX tools [4] in a series of short runs. These 360+ applications cover a wide range of functionality, such as networking tools, text editors, login utilities, archiving tools, etc. While the tests generated by KLEE during these runs are not sufficient to achieve high coverage (due to incomplete modeling), we did find many bugs quickly: 21 bugs in BUSYBOX and another 21 in MINIX have been reported (many additional reports await inspection). Figure 10 gives the command lines for the BUSYBOX bugs. All bugs were memory errors and were fixed promptly, with the exception of `date` which had been fixed in an unreleased tree. We have not heard back from the MINIX developers.

5.5 Checking tool equivalence

Thus far, we have focused on finding generic errors that do not require knowledge of a program's intended behavior. We now show how to do much deeper checking, including verifying full functional correctness on a finite set of explored paths.

KLEE makes no approximations: its constraints have perfect accuracy down to the level of a single bit. If KLEE reaches an `assert` and its constraint solver states the false branch of the `assert` cannot execute given the current path constraints, then it has *proved* that no value exists on *the current path* that could violate the assertion, modulo bugs in KLEE or non-determinism in the code. [9]

[9] Code that depends on the values of memory addresses will not

```
1 : unsigned mod_opt(unsigned x, unsigned y) {
2 :   if((y & −y) == y) // power of two?
3 :     return x & (y−1);
4 :   else
5 :     return x % y;
6 : }
7 : unsigned mod(unsigned x, unsigned y) {
8 :   return x % y;
9 : }
10: int main() {
11:   unsigned x,y;
12:   make_symbolic(&x, sizeof(x));
13:   make_symbolic(&y, sizeof(y));
14:   assert(mod(x,y) == mod_opt(x,y));
15:   return 0;
16: }
```

Figure 11: Trivial program illustrating equivalence checking. KLEE proves total equivalence when $y \neq 0$.

Importantly, KLEE will do such proofs for any condition the programmer expresses as C code, from a simple non-null pointer check, to one verifying the correctness of a program's output.

This property can be leveraged to perform deeper checking as follows. Assume we have two procedures `f` and `f'` that take a single argument and purport to implement the same interface. We can verify functional equivalence on a per-path basis by simply feeding them the same symbolic argument and asserting they return the same value: `assert(f(x) == f'(x))`. Each time KLEE follows a path that reaches this assertion, it checks if any value exists on that path that violates it. If it finds none exists, then it has proven functional equivalence on that path. By implication, if one function is correct along the path, then equivalence proves the other one is as well. Conversely, if the functions compute different values along the path and the `assert` fires, then KLEE will produce a test case demonstrating this difference. These are both powerful results, completely beyond the reach of traditional testing. One way to look at KLEE is that it automatically translates a path through a C program into a form that a theorem prover can reason about. As a result, proving path equivalence just takes a few lines of C code (the assertion above), rather than an enormous manual exercise in theorem proving.

Note that equivalence results only hold on the finite set of paths that KLEE explores. Like traditional testing, it cannot make statements about paths it misses. However, if KLEE is able to exhaust all paths then it has shown total equivalence of the functions. Although not tractable in general, many isolated algorithms can be tested this way, at least up to some input size.

We help make these points concrete using the contrived example in Figure 11, which crosschecks a trivial modulo implementation (`mod`) against one that optimizes for modulo by powers of two (`mod_opt`). It first makes the inputs `x` and `y` symbolic and then uses the `assert` (line 14) to check for differences. Two code paths reach this `assert`, depending on whether the test for power-of-two (line 2) succeeds or fails. (Along the way, KLEE generates a division-by-zero test case for when $y = 0$.) The true path uses the solver to check that the constraint $(y\& - y) == y$ implies $(x\&(y-1)) == x\%y$ holds for all values. This query succeeds. The false path checks the vacuous tautology that the constraint $(y\& - y) \neq y$ implies that $x\%y == x\%y$ also holds. The KLEE checking run then terminates, which means that KLEE has proved equivalence for all non-zero values using only a few lines of code.

This methodology is useful in a broad range of contexts. Most standardized interfaces — such as libraries, networking servers, or compilers — have multiple implementations (a partial motivation for and consequence of standardization). In addition, there are other common cases where multiple implementations exist:

1. `f` is a simple reference implementation and `f'` a real-world optimized version.
2. `f'` is a patched version of `f` that purports only to remove bugs (so should have strictly fewer crashes) or refactor code without changing functionality.
3. `f` has an inverse, which means we can change our equivalence check to verify $f^{-1}(f(x)) \equiv x$, such as: `assert(uncompress(compress(x))==x)`.

Experimental results. We show that this technique can find deep correctness errors and scale to real programs by crosschecking 67 COREUTILS tools against their allegedly equivalent BUSYBOX implementations. For example, given the same input, the BUSYBOX and COREUTILS versions of `wc` should output the same number of lines, words and bytes. In fact, both the BUSYBOX and COREUTILS tools intend to conform to IEEE Standard 1003.1 [3] which specifies their behavior.

We built a simple infrastructure to make crosschecking automatic. Given two tools, it renames all their global symbols and then links them together. It then runs both with the same symbolic environment (same symbolic arguments, files, etc.) and compares the data printed to `stdout`. When it detects a mismatch, it generates a test case that can be run to natively to confirm the difference.

Table 3 shows a subset of the mismatches found by KLEE. The first three lines show hard correctness errors (which were promptly fixed by developers), while the others mostly reveal missing functionality. As an example of a serious correctness bug, the first line gives the inputs that when run on BUSYBOX's `comm` causes it to behave as if two non-identical files were identical.

satisfy determinism since KLEE will almost certainly allocate memory objects at different addresses than native runs.

Input	BUSYBOX	COREUTILS
`comm t1.txt t2.txt`	[does not show difference]	[shows difference]
`tee -`	[does not copy twice to stdout]	[does]
`tee "" <t1.txt`	[infinite loop]	[terminates]
`cksum /`	`"4294967295 0 /"`	`"/: Is a directory"`
`split /`	`"/: Is a directory"`	
`tr`	[duplicates input on stdout]	`"missing operand"`
`[0 ``<'' 1]`		`"binary operator expected"`
`sum -s <t1.txt`	`"97 1 -"`	`"97 1"`
`tail -2l`	[rejects]	[accepts]
`unexpand -f`	[accepts]	[rejects]
`split -`	[rejects]	[accepts]
`ls --color-blah`	[accepts]	[rejects]
t1.txt: a *t2.txt:* b		

Table 3: Very small subset of the mismatches KLEE found between the BUSYBOX and COREUTILS versions of equivalent utilities. The first three are serious correctness errors; most of the others are revealing missing functionality.

Test	Random	KLEE	ELOC
With Disk	50.1%	67.1%	4617
No Disk	48.0%	76.4%	2662

Table 4: Coverage on the HISTAR kernel for runs with up to three system calls, configured with and without a disk. For comparison we did the same runs using random inputs for one million trials.

5.6 The HiStar OS kernel

We have also applied KLEE to checking non-application code by using it to check the HiStar [39] kernel. We used a simple test driver based on a user-mode HISTAR kernel. The driver creates the core kernel data structures and initializes a single process with access to a single page of user memory. It then calls the test function in Figure 12, which makes the user memory symbolic and executes a predefined number of system calls using entirely symbolic arguments. As the system call number is encoded in the first argument, this simple driver effectively tests all (sequences of) system calls in the kernel.

Although the setup is restrictive, in practice we have found that it can quickly generate test cases — sequences of system call vectors and memory contents — which cover a large portion of the kernel code and uncover interesting behaviors. Table 4 shows the coverage obtained for the core kernel for runs with and without a disk. When configured with a disk, a majority of the uncovered code can only be triggered when there are a large number of kernel objects. This currently does not happen in our testing environment; we are investigating ways to exercise this code adequately during testing. As a quick comparison, we ran one million random tests through the same driver (similar to § 5.2.4). As Table 4 shows, KLEE's tests achieve significantly more coverage than random testing both for runs with (+17.0%) and without (+28.4%) a disk.

```
1 : static void test(void *upage, unsigned num_calls) {
2 :   make_symbolic(upage, PGSIZE);
3 :   for (int i=0; i<num_calls; i++) {
4 :     uint64_t args[8];
5 :     for (int j=0; j<8; j++)
6 :       make_symbolic(&args[j], sizeof(args[j]));
7 :     kern_syscall(args[0], args[1], args[2], args[3],
8 :                  args[4], args[5], args[6], args[7]);
9 :   }
10:   sys_self_halt();
11: }
```

Figure 12: Test driver for HISTAR: it makes a single page of user memory symbolic and executes a user-specified number of system calls with entirely symbolic arguments.

KLEE's constraint-based reasoning allowed it to find a tricky, critical security bug in the 32-bit version of HISTAR. Figure 13 shows the code for the function containing the bug. The function `safe_addptr` is supposed to set `*of` to true if the addition overflows. However, because the inputs are 64 bit long, the test used is insufficient (it should be `(r < a) || (r < b)`) and the function can fail to indicate overflow for large values of `b`.

The `safe_addptr` function validates user memory addresses prior to copying data to or from user space. A kernel routine takes a user address and a size and computes if the user is allowed to access the memory in that range; this routine uses the overflow to prevent access when a computation could overflow. This bug in computing overflow therefore allows a malicious process to gain access to memory regions outside its control.

6 Related Work

Many recent tools are based on symbolic execution [11, 14–16, 20–22, 24, 26, 27, 36]. We contrast how KLEE deals with the environment and path explosion problems.

To the best of our knowledge, traditional symbolic ex-

```
1 : uintptr_t safe_addptr(int *of, uint64_t a, uint64_t b) {
2 :   uintptr_t r = a + b;
3 :   if (r < a)
4 :     *of = 1;
5 :   return r;
6 : }
```

Figure 13: HISTAR function containing an important security vulnerability. The function is supposed to set `*of` to true if the addition overflows but can fail to do so in the 32-bit version for very large values of `b`.

ecution systems [17, 18, 32] are static in a strict sense and do not interact with the running environment at all. They either cannot handle programs that make use of the environment or require a complete working model. More recent work in test generation [16, 26, 36] does allow external interactions, but forces them to use entirely concrete procedure call arguments, which limits the behaviors they can explore: a concrete external call will do exactly what it did, rather than all things it could potentially do. In KLEE, we strive for a functional balance between these two alternatives; we allow both interaction with the outside environment and supply a model to simulate interaction with a symbolic one.

The path explosion problem has instead received more attention [11, 22, 24, 27, 34]. Similarly to the search heuristics presented in Section 3, search strategies proposed in the past include Best First Search [16], Generational Search [27], and Hybrid Concolic Testing [34]. Orthogonal to search heuristics, researchers have addressed the path explosion problem by testing paths compositionally [8, 24], and by tracking the values read and written by the program [11].

Like KLEE, other symbolic execution systems implement their own optimizations before sending the queries to the underlying constraint solver, such as the simple syntactic transformations presented in [36], and the *constraint subsumption* optimization discussed in [27].

Similar to symbolic execution systems, model checkers have been used to find bugs in both the design and the implementation of software [10, 12, 19, 25, 29, 30]. These approaches often require a lot of manual effort to build test harnesses. However, the approaches are somewhat complementary to KLEE: the tests KLEE generates can be used to drive the model checked code, similar to the approach embraced by Java PathFinder [31, 37].

Previously, we showed that symbolic execution can find correctness errors by crosschecking various implementations of the same library function [15]; this paper shows that the technique scales to real programs. Subsequent to our initial work, others applied similar ideas to finding correctness errors in applications such as network protocol implementations [13] and PHP scripts [9].

7 Conclusion

Our long-term goal is to take an arbitrary program and routinely get 90%+ code coverage, crushing it under test cases for all interesting inputs. While there is still a long way to go to reach this goal, our results show that the approach works well across a broad range of real code. Our system KLEE, automatically generated tests that, on average, covered over 90% of the lines (in aggregate over 80%) in roughly 160 complex, system-intensive applications "out of the box." This coverage significantly exceeded that of their corresponding hand-written test suites, including one built over a period of 15 years.

In total, we used KLEE to check 452 applications (with over 430K lines of code), where it found 56 serious bugs, including ten in COREUTILS, arguably the most heavily-tested collection of open-source applications. To the best of our knowledge, this represents an order of magnitude more code and distinct programs than checked by prior symbolic test generation work. Further, because KLEE's constraints have no approximations, its reasoning allow it to prove properties of paths (or find counter-examples without false positives). We used this ability both to prove path equivalence across many real, purportedly identical applications, and to find functional correctness errors in them.

The techniques we describe should work well with other tools and provide similar help in handling a broad class of applications.

8 Acknowledgements

We thank the GNU COREUTILS developers, particularly the COREUTILS maintainer Jim Meyering for promptly confirming our reported bugs and answering many questions. We similarly thank the developers of BUSYBOX, particularly the BUSYBOX maintainer, Denys Vlasenko. We also thank Nickolai Zeldovich, the designer of HISTAR, for his great help in checking HISTAR, including writing a user-level driver for us. We thank our shepard Terence Kelly, the helpful OSDI reviewers, and Philip Guo for valuable comments on the text. This research was supported by DHS grant FA8750-05-2-0142, NSF TRUST grant CCF-0424422, and NSF CAREER award CNS-0238570-001. A Junglee Graduate Fellowship partially supported Cristian Cadar.

References

[1] Busybox. `www.busybox.net`, August 2008.

[2] Coreutils. `www.gnu.org/software/coreutils`, August 2008.

[3] IEEE Std 1003.1, 2004 edition. `www.unix.org/version3/ieee_std.html`, May 2008.

[4] MINIX 3. `www.minix3.org`, August 2008.

[5] SecurityFocus, `www.securityfocus.com`, March 2008.

[6] uCLibc. `www.uclibc.org`, May 2008.

[7] United States National Vulnerability Database, nvd.nist.gov, March 2008.

[8] ANAND, S., GODEFROID, P., AND TILLMANN, N. Demand-driven compositional symbolic execution. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*.

[9] ARTZI, S., KIEŻUN, A., DOLBY, J., TIP, F., DIG, D., PARADKAR, A., AND ERNST, M. D. Finding bugs in dynamic web applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2008)*.

[10] BALL, T., AND RAJAMANI, S. Automatically validating temporal safety properties of interfaces. In *SPIN 2001 Workshop on Model Checking of Software* (May 2001).

[11] BOONSTOPPEL, P., CADAR, C., AND ENGLER, D. RWset: Attacking path explosion in constraint-based test generation. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*.

[12] BRAT, G., HAVELUND, K., PARK, S., AND VISSER, W. Model checking programs. In *IEEE International Conference on Automated Software Engineering (ASE 2000)*.

[13] BRUMLEY, D., CABALLERO, J., LIANG, Z., NEWSOME, J., AND SONG, D. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of USENIX Security Symposium (USENIX Security 2007)*.

[14] BRUMLEY, D., NEWSOME, J., SONG, D., WANG, H., AND JHA, S. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (IEEE S&P 2006)*.

[15] CADAR, C., AND ENGLER, D. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN 2005)*.

[16] CADAR, C., GANESH, V., PAWLOWSKI, P., DILL, D., AND ENGLER, D. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006)*.

[17] CLARKE, E., AND KROENING, D. Hardware verification using ANSI-C programs as a reference. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC 2003)*.

[18] CLARKE, E., KROENING, D., AND LERDA, F. A tool for checking ANSI-C programs. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*.

[19] CORBETT, J., DWYER, M., HATCLIFF, J., LAUBACH, S., PASAREANU, C., ROBBY, AND ZHENG, H. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the International Conference on Software Engineering (ICSE 2000)*.

[20] COSTA, M., CASTRO, M., ZHOU, L., ZHANG, L., AND PEINADO, M. Bouncer: Securing software by blocking bad input. In *Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP 2007)*.

[21] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: end-to-end containment of Internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*.

[22] EMMI, M., MAJUMDAR, R., AND SEN, K. Dynamic test input generation for database applications. In *International Symposium on Software Testing and Analysis (ISSTA 2007)*.

[23] GANESH, V., AND DILL, D. L. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*.

[24] GODEFROID, P. Compositional dynamic test generation. In *Proceedings of the 34th Symposium on Principles of Programming Languages (POPL 2007)*.

[25] GODEFROID, P. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL 1997)*.

[26] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed automated random testing. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI 2005)*.

[27] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated whitebox fuzz testing. In *Proceedings of Network and Distributed Systems Security (NDSS 2008)*.

[28] HOFFMANN, J., AND KOEHLER, J. A new method to index and query sets. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI 1999)*.

[29] HOLZMANN, G. J. From code to models. In *Proceedings of 2nd International Conference on Applications of Concurrency to System Design (ACSD 2001)*.

[30] HOLZMANN, G. J. The model checker SPIN. *Software Engineering 23*, 5 (1997), 279–295.

[31] KHURSHID, S., PASAREANU, C. S., AND VISSER, W. Generalized symbolic execution for model checking and testing. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*.

[32] KROENING, D., CLARKE, E., AND YORAV, K. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th Design Automation Conference (DAC 2003)*.

[33] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization (CGO 2004)*.

[34] MAJUMDAR, R., AND SEN, K. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*.

[35] MILLER, B., KOSKI, D., LEE, C. P., MAGANTY, V., MURTHY, R., NATARAJAN, A., AND STEIDL, J. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Tech. rep., University of Wisconsin - Madison, 1995.

[36] SEN, K., MARINOV, D., AND AGHA, G. CUTE: A concolic unit testing engine for C. In *In 5th joint meeting of the European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*.

[37] VISSER, W., PASAREANU, C. S., AND KHURSHID, S. Test input generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*.

[38] YANG, J., SAR, C., AND ENGLER, D. eXplode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*.

[39] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*.

Hardware Enforcement of Application Security Policies Using Tagged Memory

Nickolai Zeldovich*, Hari Kannan†, Michael Dalton†, and Christos Kozyrakis†
*MIT †Stanford University

ABSTRACT

Computers are notoriously insecure, in part because application security policies do not map well onto traditional protection mechanisms such as Unix user accounts or hardware page tables. Recent work has shown that application policies can be expressed in terms of information flow restrictions and enforced in an OS kernel, providing a strong assurance of security. This paper shows that enforcement of these policies can be pushed largely into the processor itself, by using tagged memory support, which can provide stronger security guarantees by enforcing application security even if the OS kernel is compromised.

We present the *Loki* tagged memory architecture, along with a novel operating system structure that takes advantage of tagged memory to enforce application security policies in hardware. We built a full-system prototype of Loki by modifying a synthesizable SPARC core, mapping it to an FPGA board, and porting HiStar, a Unix-like operating system, to run on it. One result is that Loki allows HiStar, an OS already designed to have a small trusted kernel, to further reduce the amount of trusted code by a factor of two, and to enforce security despite kernel compromises. Using various workloads, we also demonstrate that HiStar running on Loki incurs a low performance overhead.

1 INTRODUCTION

A significant part of the computer security problem stems from the fact that security of large-scale applications usually depends on millions of lines of code behaving correctly, rendering security guarantees all but impossible. One way to improve security is to separate the enforcement of security policies into a small, trusted component, typically called the trusted computing base [19], which can then ensure security even if the other components are compromised. This usually means enforcing security policies at a lower level in the system, such as in the operating system or in hardware. Unfortunately, enforcing application security policies at a lower level is made difficult by the *semantic gap* between different layers of abstraction in a system. Since the interface traditionally provided by the OS kernel or by hardware is not expressive enough to capture the high-level semantics of application security policies, applications resort to building their own ad-hoc security mechanisms. Such mechanisms are often poorly designed and implemented, leading to an endless stream of compromises [22].

As an example, consider a web application such as Facebook or MySpace, where the web server stores personal profile information for millions of users. The application's security policy requires that one user's profile can be sent only to web browsers belonging to the friends of that user. Traditional low-level protection mechanisms, such as Unix's user accounts or hardware's page tables, are of little help in enforcing this policy, since they were designed with other policies in mind. In particular, Unix accounts can be used by a system administrator to manage different users on a single machine; Unix processes can be used to provide isolation; and page tables can help in protecting the kernel from application code. However, enforcing or even expressing our example website's high-level application security policy using these mechanisms is at best difficult and error-prone [17]. Instead, such policies are usually enforced throughout the application code, effectively making the entire application part of the trusted computing base.

A promising technique for bridging this semantic gap between security mechanisms at different abstraction layers is to think of security in terms of what can happen to data, instead of specifying the individual operations that can be invoked at any particular layer (such as system calls). For instance, recent work on operating systems [10, 18, 35, 36] has shown that many application security policies can be expressed as restrictions on the movement of data in a system, and that these security policies can then be enforced using an information flow control mechanism in the OS kernel.

This paper shows that hardware support for tagged memory allows enforcing data security policies at an even lower level—directly in the processor—thereby providing application security guarantees even if the kernel is compromised. To support this claim, we designed Loki, a hardware architecture that provides a word-level memory tagging mechanism, and ported the HiStar operating system [35] (which was designed to enforce application security policies in a small trusted kernel) to run on Loki. Loki's tagged memory simplifies security enforcement by associating security policies with data at the lowest level in the system—in physical memory. The

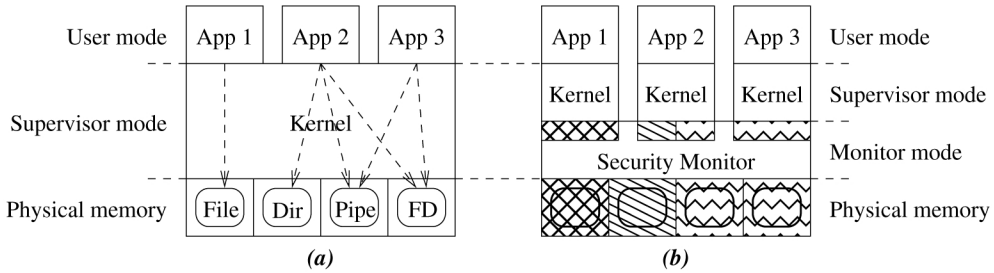


Figure 1: A comparison between (a) traditional operating system structure, and (b) this paper's proposed structure using a security monitor. Horizontal separation between application boxes in (a), and between stacks of applications and kernels in (b), indicates different protection domains. Dashed arrows in (a) indicate access rights of applications to pages of memory. Shading in (b) indicates tag values, with small shaded boxes underneath protection domains indicating the set of tags accessible to that protection domain.

resulting simplicity is evidenced by the fact that the port of HiStar to Loki has less than half the amount of trusted code than HiStar running on traditional CPUs. Finally, we show that tagged memory can achieve strong security guarantees at a minimal performance cost, by building and evaluating a full system prototype of Loki running HiStar.

While a tagged memory mechanism on its own can control read and write access to physical resources, it is not sufficient for enforcing strict information flow control. In particular, the lack of a name translation mechanism makes it difficult to avoid certain kinds of covert channels, as we will discuss later. To this end, this paper presents a novel OS structure that can enforce the same application security policy under two threat models. The first is a simpler *discretionary access control* model, which aims to control read and write access to memory, and is enforced largely in hardware. The second is a more complex *mandatory access control* model, which aims to control all ways in which data could be passed between processes, and is enforced in an OS kernel. The key difference between our OS structure and a traditional one is that the kernel is not trusted to enforce the discretionary parts of its mandatory access control model. Instead, it is the hardware's job to control read and write access to memory, and the kernel is only trusted to minimize covert channels.

The rest of the paper is structured as follows. The next section describes our overall system architecture and its security goals, as well as our experimental prototype. Section 3 describes the structure of our operating system in more detail, and Section 4 describes the tagged memory processor we developed as part of this work. Section 5 presents an evaluation of the security and performance of our prototype, Section 6 discusses related work, and Section 7 concludes.

2 System architecture

This paper describes a combination of a new hardware architecture, called Loki, that enforces security policies in hardware by using tagged memory, together with a modified version of the HiStar operating system [35], called LoStar, that enforces discretionary access components of its information flow policies using Loki. The overall structure of this system is shown in Figure 1.

Traditional OS kernels, shown in Figure 1 (a), are tasked with both implementing abstractions seen by user-level code as well as controlling access to data stored in these abstractions. LoStar, shown in Figure 1 (b), separates these two functions by using hardware to control data access. In particular, the Loki hardware architecture associates *tags* with words of memory, and allows specifying protection domains in terms of the tags that can be accessed. LoStar manages these tags and protection domains from a small software component, called the *security monitor*, which runs underneath the kernel in a special processor privilege mode called *monitor mode*. The security monitor translates application security policies on data, specified in terms of *labels* on kernel objects in the HiStar operating system, into tags on the corresponding physical memory, which the hardware then enforces.

Most systems enforce security policies in hardware through a translation mechanism, such as paging or segmentation. However, enforcing security in a translation mechanism means that security policies are bound to virtual resources, and not to the actual physical memory storing the data being protected. As a result, the policy for a particular piece of data in memory is not well-defined in hardware, and instead depends on various invariants being implemented correctly in software, such as the absence of aliasing. Tagging physical memory helps bridge the semantic gap between the data and its security policy, and makes the security policy unambiguous even at a low level, while requiring a much smaller trusted code base.

As mentioned previously, tagged memory alone is not sufficient for enforcing strict information flow control, because dynamic allocation of resources with fixed names, such as physical memory, contains inherent covert channels. For example, a malicious process with access to a secret bit of data could signal that bit to a

colluding non-secret process on the same machine by allocating many physical memory pages and freeing only the odd- or even-numbered pages depending on the bit value. Operating systems like HiStar solve such problems by virtualizing resource names (e.g. using kernel object IDs) and making sure that these virtual names are never reused. However, the additional kernel complexity can lead to bugs far worse than the covert channels the added code was trying to fix. Moreover, implementing equivalent functionality in hardware would not be inherently any simpler than the OS kernel code it would be replacing, and would not necessarily improve security.

What hardware support for tagged memory can address, however, is the the tension between stronger security and increased complexity seen in an OS kernel. In particular, this paper introduces a new, intermediate level of security provided by hardware, which can enforce a subset of the kernel's security guarantees, as illustrated by our hybrid threat model in Figure 2. In the simplest case, we are concerned with two security levels, *high* and *low*, and the goal is ensuring that data from the high level cannot influence data in the low level. There are multiple interpretations of high and low. For instance, high might represent secret user data, in which case low would be world-readable, as in [2]. Alternatively, low could represent integrity-protected system configuration files, which should not be affected by high user inputs, as in [3].

The hybrid model provides different enforcement of our security goal under different assumptions. In particular, the weaker *discretionary access control* model, enforced by the tagging hardware and the security monitor, disallows both high processes from modifying low data and low processes from reading high data. However, if a malicious pair of high and low processes collude, they can exploit covert channels to subvert our security goal, as shown by the dashed arrow in Figure 2. The stronger *mandatory access control* model aims to prevent such covert communication, by providing a carefully designed kernel interface, like the one in HiStar, in a more complex OS kernel. The resulting hybrid model can enforce security largely in hardware in the case of only one malicious or compromised process, and relies on the more complex OS kernel when there are multiple malicious processes that are colluding.

The rest of this section will first describe LoStar from the point of view of different applications, illustrating the security guarantees provided by different parts of the operating system. We will then provide an overview of the Loki hardware architecture, and discuss how the LoStar operating system uses Loki's hardware mechanisms.

2.1 Application perspective

One example of an application in LoStar is the Unix environment itself. HiStar implements Unix in a user-space

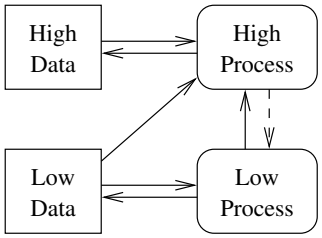


Figure 2: A comparison of the *discretionary access control* and *mandatory access control* threat models. Rectangles represent data, such as files, and rounded rectangles represent processes. Arrows indicate permitted information flow to or from a process. A dashed arrow indicates information flow permitted by the discretionary model but prohibited by the mandatory model.

library, which in turn uses HiStar's kernel labels to implement its protection, such as the isolation of a process's address space, file descriptor sharing, and file system access control. As a result, unmodified Unix applications running on LoStar do not need to explicitly specify labels for any of their objects. The Unix library automatically specifies labels that mimic the security policies an application would expect on a traditional Unix system. However, even the Unix library is not aware of the translation between labels and tags being done by the kernel and the security monitor. Instead, the kernel automatically passes the label for each kernel object to the underlying security monitor.

LoStar's security monitor, in turn, translates these labels into tags on the physical memory containing the respective data. As a result, Loki's tagged memory mechanism can directly enforce Unix's discretionary security policies without trusting the kernel. For example, a page of memory representing a file descriptor is tagged in a way that makes it accessible only to the processes that have been granted access to that file descriptor. Similarly, the private memory of a process's address space can be tagged to ensure that only threads within that particular process can access that memory. Finally, Unix user IDs are also mapped to labels, which are then translated into tags and enforced using the same hardware mechanism.

An example of an application that relies on both discretionary and mandatory access control is the HiStar web server [36]. Unlike other Unix applications, which rely on the Unix library to automatically specify all labels for them, the web server explicitly specifies a different label for each user's data, to ensure that user data remains private even when handled by malicious web applications. In this case, if an attacker cannot compromise the kernel, user data privacy is enforced even when users invoke malicious web applications on their data. On the other hand, if an attacker can compromise the kernel, malicious web applications can leak private data from one user to another, but only for users that invoke the malicious code. Users that don't invoke the malicious code

will still be secure, as the security monitor will not allow malicious kernel code to access arbitrary user data.

2.2 Hardware overview

The design of the Loki hardware architecture was driven by three main requirements. First, hardware should provide a large number of non-hierarchical protection domains, to be able to express application security policies that involve a large number of disjoint principals. Second, the hardware protection mechanism should protect low-level physical resources, such as physical memory or peripheral devices, in order to push enforcement of security policies to the lowest possible level. Finally, practical considerations require a fine-grained protection mechanism that can specify different permissions for different words of memory, in order to accommodate programming techniques like the use of contiguous data structures in C where different data structure members could have different security properties.

To address these requirements, Loki logically associates an opaque 32-bit *tag* with every 32-bit word of physical memory. Tag values correspond to a security policy on the data stored in locations with that particular tag. Protection domains in Loki are specified in terms of tags, and can be thought of as a mapping between tags and permission bits (read, write, and execute). Loki provides a software-filled *permissions cache* in the processor, holding permission bits for some set of tags accessed by the current protection domain, which is checked by the processor on every instruction fetch, load, and store.

A naive implementation of word-level tags could result in a 100% memory overhead for tag storage. To avoid this problem, Loki implements a multi-granular tagging scheme, which allows tagging an entire page of memory with a single 32-bit tag value. This optimization turns out to be quite effective, and will be described in more detail later in the paper.

Tag values and permission cache entries can only be updated in Loki while in a special processor privilege mode called *monitor mode*, which can be logically thought of as more privileged than the traditional supervisor processor mode. Hardware invokes tag handling code running in monitor mode on any tag permission check failure or permission cache miss by raising a *tag exception*. To avoid including page table handling code in the trusted computing base, the processor's MMU is disabled while executing in monitor mode.

2.3 OS overview

Kernel code in Loki continues to execute at the supervisor privilege level, with access to all existing privileged supervisor instructions. This includes access to traditionally privileged state, such as control registers, the MMU, page tables, and so on. However, kernel code does not have direct access to instructions that modify tags or permission cache entries. Instead, it invokes the security monitor to manage the tags and the permission cache, subject to security checks that we will describe later.

The kernel requires word-level tags for two main reasons. First, existing C data structures often combine data with different security requirements in contiguous memory. For example, the security label field in a kernel object should not be writable by kernel code, but the rest of the object's data can be made writable, subject to the policy specified by the security label. Word-level tagging avoids the need to split up such data structures into multiple parts according to security requirements. Second, word-level tags reduce the overhead of placing a small amount of data, such as a 32-bit pointer or a 64-bit object ID, in a unique protection domain.

Although Loki enforces memory access control, it does not guarantee liveness. All of the kernel protection domains in LoStar participate in a cooperative scheduling protocol, explicitly yielding the CPU to the next protection domain when appropriate. Buggy or malicious kernel code can perform a denial of service attack by refusing to yield, yielding only to other colluding malicious kernels, halting the processor, misconfiguring interrupts, or entering an infinite loop. Liveness guarantees can be enforced at the cost of a larger trusted monitor, which would need to manage timer interrupts, perform preemptive scheduling, and prevent processor state corruption.

3 Operating system design

To illustrate how Loki can be used to minimize the amount of trusted code, we modified HiStar, an operating system designed to minimize the amount of trusted code, to take advantage of tags to enforce its security guarantees in a smaller TCB. The rest of this section first motivates our choice of the HiStar operating system, then provides a brief overview of HiStar, and finally describes the modifications required to port HiStar to Loki in detail.

3.1 OS choice rationale

Enforcing application security policies at a low level requires addressing two main problems. First, applications must be able to express their security policies to the underlying system in a uniform manner, so that their policies can then be enforced, and second, application-level names, like filenames, must be securely bound to low-level protection domains, like memory tag values.

Traditional Unix-like operating systems are not a particularly good fit for addressing these two problems. Unix provides a large number of protection mechanisms, from process isolation to file descriptor sharing to user IDs, which have poorly defined semantics [5] and are

cumbersome to use in practice for building secure applications [17]. At the same time, mapping Unix filenames to the underlying object (inode) and its protection domain involves many layers of translation in kernel code. All of this kernel code must be fully trusted, since any mis-translation can subvert the intent of a privileged application by causing it to access an arbitrary file or device.

HiStar was an appealing choice for this work because it addressed both of these problems. First, HiStar used a single kernel mechanism—information flow control—to implement all protection in the system, from emulating Unix security to expressing application security policies. This meant that extending the enforcement of this single mechanism into hardware would automatically enforce all higher-level security policies implemented using HiStar's protection mechanism. Second, as we will discuss later on, HiStar reduces all naming to a single flat object ID space managed by the kernel. This means that a secure binding between names and protection domains can be implemented by just providing this simple namespace in the trusted security monitor.

3.2 HiStar overview

HiStar's information flow control mechanism revolves around three key concepts. The first is the notion of a *category*—an opaque 61-bit ID managed by the kernel—which represents a particular kind of data in a system, and can restrict how that data can be accessed or modified. For example, a separate category is allocated for every process, to ensure that only threads in that process can access that process's address space. A separate category is also allocated for each file descriptor to control what processes are allowed to access it. Finally, Unix user accounts are also represented with categories that mirror the user's UID.

The second notion is that of a *label*, which is a set of categories. Every kernel object has a label associated with it, and the contents of an object is subject to the restrictions of every category in that object's label.

The final notion is that of thread *ownership* of categories, which defines threads that have access to data labeled in a particular fashion. For example, every thread typically has ownership of the category corresponding to its process, categories for any file descriptors it has access to, and the category of the Unix user on whose behalf the process is executing.

HiStar reduces the amount of trusted kernel code compared to traditional OSes by providing a simple, low-level kernel interface, consisting of six kernel object types: *segments*, *address spaces*, *devices*, *threads*, *gates*, and *containers*. Kernel objects are named by 61-bit object IDs that are unique over all time, and most application-level naming is reduced to the kernel's trusted object ID namespace. For example, Unix process IDs are object IDs of the container object representing the process. Pseudo-terminal (pty) IDs correspond to the object ID of a segment object storing that pseudo-terminal's control block. Even file and directory inodes correspond to object IDs of the segments and containers used to implement them, and the kernel's single-level store provides persistent disk storage.

The two kernel object types of particular interest in this paper are threads and gates. *Thread* objects are used to execute user-level code, and consist of a register set and the object ID of an address space object that defines the virtual address space for the corresponding process. A thread's label reflects the data that the thread could have potentially observed. Threads can dynamically adjust their label to observe secret data at runtime. By doing so, a thread gives up the right to modify any objects not also labeled with the secret data's category, thus transitively controlling information flow. However, a thread can only add restrictions to its label, not remove them. To ensure that threads cannot unilaterally read all secret data in the system by adjusting their labels, each thread has a *clearance*, which is a set of categories that a thread is allowed to add to its label. The thread's clearance serves to enforce a form of discretionary access control.

Gate objects provide a mechanism for protected control transfer, allowing a thread to switch to a particular entry point in another address space and protection domain. Gates can be thought of as an IPC mechanism, except that the client, instead of the server, provides the initial thread of execution. The gate's privileges are stored in the *label* and *clearance* associated with the gate.

The kernel provides a small number of operations (system calls) that can be performed on each type of kernel object by threads. For each operation, the kernel knows how information can flow as a result of the operation. Whenever a thread asks the kernel to perform an operation on another object, the kernel compares the thread's label to the label of the other object, and decides whether the labels allow the operation.

3.3 Minimizing trusted code

HiStar's design already provides a significantly smaller fully trusted kernel than a traditional Unix system, as shown in Figure 3 (a) and (b). Code implementing traditional Unix semantics is moved to an untrusted user-level library, while security policies, specified by either the Unix library or the application in terms of labels, are enforced by a much smaller kernel.

The Loki architecture allows us to further reduce HiStar's trusted code base, by enforcing a subset of HiStar's security guarantees with a small *security monitor* in a system called LoStar, as shown in Figure 3 (c). At a high level, the kernel in LoStar still enforces *information flow*

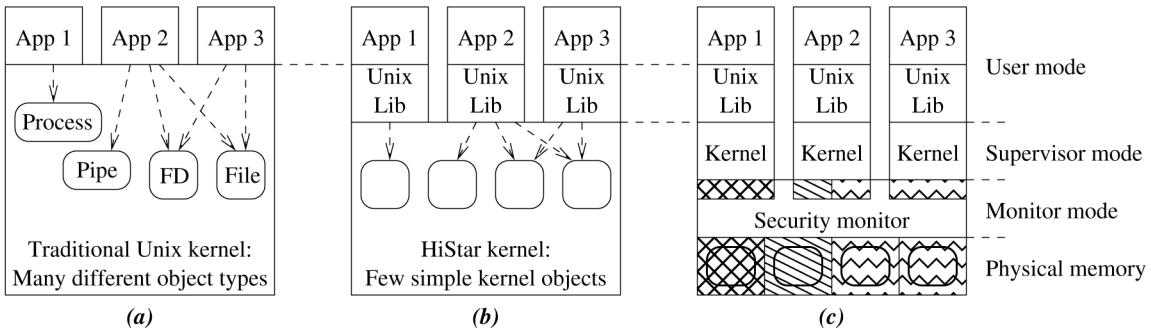


Figure 3: A comparison of operating system structure, showing (a) traditional Unix, (b) HiStar, and (c) LoStar. Vertical stacks correspond to different protection domains. The Unix library and kernel components in different protection domains in (c) are executing the same code with different privileges, similar to how Unix processes execute the same `libc` code with different privileges.

control: if a malicious application gains access to secret data, the kernel is responsible for ensuring that the malicious code cannot export this data outside of the system. However, the security monitor enforces a simpler *discretionary access control* policy, ensuring that objects can only be read or written by a thread whose label allows that operation, and ensuring that threads can only change their labels in approved ways (that is, not allowing a thread to arbitrarily lower its label or to raise it above its clearance). This effectively translates to enforcing Unix security policies that prevent one user's process from reading another user's files.

In LoStar, the kernel is no longer a single entity with a fixed set of privileges. Rather, there is a separate logical instance of the kernel associated with each running thread, and the kernel instance derives its privileges from the associated thread. (In HiStar, the kernel only has a notion of a thread, which has its own set of privileges, although multiple threads in the same Unix process, as implemented by the Unix library, often have the same privileges.) Moreover, all shared state in the system is managed by the security monitor, which means that one kernel instance cannot directly compromise another kernel instance by corrupting its data structures. As a result, the kernel can be viewed as just a library (much like libc, where the OS kernel maintains and protects all of libc's data), which means there is no longer any notion of a global kernel compromise. Instead, LoStar resembles a distributed system, in which many kernel instances cooperate in limited ways via the security monitor's mechanisms. The security monitor, in turn, protects different kernel instances from one another by write-protecting all kernel text and read-only data, and facilitates controlled sharing between kernels by allowing each kernel to explicitly specify labels to control read-write access to kernel objects.

HiStar's security policies, represented by labels on kernel objects, ultimately come from applications, often with the help of the Unix library. The security monitor maintains a mapping between these HiStar labels and Loki *tags*, and tags all physical pages of memory belonging to each kernel object with the tag corresponding to that object's label. This mapping is a fundamental part of the design, as hardware's fixed-width tag values cannot directly represent HiStar's variable-size labels. Our prototype uses the label's 32-bit physical address as the tag value. To ensure that tag values always refer to valid labels, the security monitor keeps a tag reference count for each label object, and avoids garbage collecting any referenced label. The monitor also keeps track of the Loki *permission cache* or P-cache for each thread's protection domain, and loads these permissions into the hardware cache on each context switch.

When the application or kernel code, running outside of monitor mode, accesses data with a particular tag for the first time, hardware raises a *tag exception*, which traps to the monitor. The monitor then looks up the HiStar label corresponding to the accessed tag value, compares it to the currently executing thread's HiStar label, and updates the permission cache accordingly.

Our design requires that applications be able to securely specify labels to the underlying kernel, which in turn relays them to the security monitor. While a compromised kernel instance could modify a label specified by an application to change the effective security policy, our design mitigates such attacks by treating different instances of the kernel as independent libraries that can only affect each other through the protected interface provided by the security monitor. As a result, our design trusts the kernel much in the same way that a traditional OS trusts libc to relay the application's security policy from the application to the kernel. More specifically, even if an attacker process were to compromise their instance of the kernel, this would not compromise other kernel instances, since the kernel code is write-protected by the security monitor, and all data structures shared by the kernels are maintained by the security monitor as well. Our design also achieves a form of "forward security": even if an attacker can compromise another process and its underlying kernel, and fully control its future

execution, it cannot change any labels that were already specified to the security monitor in the past.

3.4 Monitor functionality

Although most policies enforced by the LoStar security monitor translate directly into tags on physical memory, there are a few other guarantees that the monitor must provide which cannot be directly expressed with memory tagging. In particular, there are certain data structures, such as object labels, reference counts, and global hash tables, that should not be modified arbitrarily by untrusted kernel code. Instead, the security monitor protects these data structures by making the relevant fields read-only to application and kernel code, and providing a system-call-like interface for modifying these fields in a safe manner, as we will now describe.

First, the monitor protects object labels, which encode the security policies in our system, by using Loki's fine-grained tags. Each kernel object includes a pointer to a label object that describes the object's security policy. When a kernel object is allocated, the monitor sets the tag value for the object's label pointer, and for all words comprising the label object, to a special tag value that is readable but not writable by all kernel code. This allows kernel code to make its own access control decisions based on an object's label, but prevents potentially compromised kernel code from subverting the security by modifying labels.

Most of the state in a HiStar system resides in kernel objects, which have well-defined labels. However, the HiStar kernel also maintains one global data structure whose integrity is crucial for controlled sharing between mutually-distrustful kernels in different protection domains. This data structure is the object hash table, which maps object IDs to kernel objects, and it is implemented using chaining, so that each kernel object has a pointer to the next object in the same hash bucket.

The monitor ensures the integrity of the object ID to kernel object mapping by tagging the hash table structure, and the linked list pointer and object ID fields in every kernel object with a special *kernel-object* tag value. This tag value allows read but not write access for all protection domains. Loki's support for fine-grained word-level tagging simplifies the enforcement of kernel data structure integrity in this case. The integrity of the object ID mapping ensures that a user-level application, which uses object IDs to access its objects, will always access the correct object. An attacker that compromises the kernel running with different access rights will not be able to substitute other objects with the same object ID into the hash table.

The object type is also protected by the monitor, to ensure that one type of object cannot be mistaken for another. The contents of different types of kernel objects have different meanings, and the monitor associates certain privileges with thread and gate objects, based on their label. If the object's type was not protected, an attacker might be able to convince the security monitor to interpret the bit-level representation of a gate object as a thread with the privileges of the original gate object. Depending on the exact bit-level layout of these objects, this might result in the monitor executing arbitrary code in a thread with the gate's privileges. Enforcing the integrity of an object's type prevents these kinds of attacks.

The monitor also protects each object's reference count field, to ensure that a malicious kernel cannot deallocate a kernel object unless it controls every reference to that object. Every reference to a kernel object corresponds to a 64-bit memory location which holds the 61-bit ID of that kernel object. These references are protected by setting the low bit, called the *reference-holder* bit, of that memory location's tag to 1 (typically tags have the low bits set to 0, since tag values are page-aligned physical memory addresses of label objects). The monitor will only increment or decrement reference counts for an object if it also atomically sets or clears the *reference-holder* bit in the tag of a 64-bit word of memory storing that object ID. Thus, memory locations tagged as *reference-holders* are effectively capabilities to un-reference that object later on. To ensure these capabilities are not tampered with, the monitor only allows read access to tags with the *reference-holder* bit set, even if the tag with that bit cleared would have allowed write access.

Other linked lists of objects in the kernel, such as lists of waiting threads, are not protected by the security monitor in the same way as the global object hash table. Instead, kernels are free to arbitrarily manipulate all pointers in such linked lists. However, well-behaved kernel code can ensure that it is traversing a valid list of kernel objects by verifying that linked list pointers point to memory with the special *kernel-object* tag, and by verifying the object's type value at each step. Although malicious kernel code can corrupt the linked list and form an infinite loop, it cannot trick another kernel into accessing a kernel object of the wrong type, or any other piece of memory that is not a kernel object, in its traversal of a linked list. In the case of a list of waiting threads, this can result in lost or spurious wakeups, or more generally, denial of service, but not data corruption.

Finally, the monitor provides a narrow interface to perform a small number of operations on these integrity-protected data structures, which we describe in the next subsection. To provide this system-call-like interface, the monitor allocates a *monitor-call* tag that is not accessible to kernel code, and a special *monitor-call* memory word, tagged with the *monitor-call* tag value, which is used to invoke the monitor. When kernel code wants to

invoke a privileged monitor operation, it places its request in its registers and accesses the *monitor-call* word. This causes a tag exception, invoking monitor code. The monitor performs the requested operation, subject to security checks, and resumes kernel execution at the next instruction, skipping the memory access that caused the exception.

3.5 Monitor call API

The monitor call interface consists of a number of operations that cannot be safely implemented through memory tagging alone, which we will now describe. The first set of operations context-switch to a different protection domain:

- Switch to another thread, represented by a kernel thread object. LoStar implements cooperative scheduling between kernels. The monitor ensures the validity of the thread object, and loads the access rights associated with that thread object before executing its kernel code.
- Invoke the garbage collection code for a particular kernel object. The monitor conceptually keeps an idle protection domain associated with each kernel object, created when a kernel object is allocated, ready to garbage collect the kernel object once its reference count reaches zero. This protection domain has implicit rights over its respective kernel object and any reference counts that this object in turn holds.
- Call a function in a special protection domain, used for the page allocator. The monitor provides a fixed-depth stack for storing the caller's protection domain and execution state while the called protection domain executes (e.g. allocating or freeing a page of memory in the page allocator). The monitor provides the called protection domain with a fresh execution stack.
- Return from a cross-domain function call, passing the return values to the caller and restoring the caller's protection domain and execution state.

The monitor also provides operations to manipulate memory, such as pages and kernel objects:

- Change the tag for a range of memory, used to transfer memory between protection domains. Any protection domain that has read and write access to a range of memory can ask the monitor to assign any other non-reserved tag (that is, not *reference-holder*, *kernel-object*, and so on) to that range of memory. The page allocator is implemented as just another protection domain; allocating memory involves the allocator re-labeling one of its free pages with the caller's requested tag value, and freeing a tag involves re-labeling a page of memory with the allocator's tag value.
- Allocate a new kernel object with a particular type, label, and clearance. The monitor allocates an empty kernel object with a fresh object ID, places it on the object hash table, and returns the object pointer to the caller for further initialization. The monitor ensures that the label (and clearance, for threads and gates) of the new object is permitted for the currently executing thread.
- Atomically increment or decrement the refcount of a kernel object and, correspondingly, set-from-zero or clear-from-one the *reference-holder* bit in the tag of a 64-bit memory location storing the object ID of that kernel object. The monitor checks that the caller has read and write privileges over the tag of the memory location with the *reference-holder* bit cleared, and to avoid potential ID splicing attacks, disallows 64-bit memory locations that span pages.

Finally, the monitor provides operations to manipulate protection domains:

- Change the protection domain of the current thread by invoking a gate. The monitor verifies the validity of the supplied gate object, and checks that the caller is authorized to invoke the gate.
- Change the protection domain of the current thread by adjusting the label or clearance, as long as it is permitted by the thread's current label and clearance.
- Allocate a new category. The monitor grants ownership of the newly allocated category to the requesting thread.

The LoStar prototype incurs some performance overhead due to the introduction of a security monitor. In particular, any communication between two instances of the kernel running in different protection domains must now go through the security monitor, and the security monitor must be involved in the creation of new protection domains, as well as switches between protection domains. Section 5 will present a more detailed evaluation of the performance overheads incurred by the introduction of the security monitor.

3.6 Interrupts

LoStar must deal with two kinds of exceptions: traditional CPU exceptions, which include synchronous processor faults and asynchronous device interrupts, and Loki's tag exceptions, which will be described in more

detail in Section 4.3. The two exception mechanisms are independent, in that the CPU maintains two separate vector tables for the two kinds of exceptions, and only tag exceptions switch the processor into monitor mode.

Traditional synchronous CPU exceptions, such as page faults or divides by zero, are handled by the currently running kernel instance in LoStar, without switching into a different protection domain. Asynchronous device interrupts are also initially vectored to the currently running kernel instance, and can either be handled by the same kernel, or handed off to the device driver's protection domain. In the case of timer interrupts, the currently running kernel instance simply runs the scheduler, which picks another thread to execute and asks the monitor to switch to that thread's kernel instance. Network device interrupts, on the other hand, are handled by invoking the security monitor to pass the interrupt to the network device driver.

To simplify the security monitor, tag exceptions mask external interrupts when transitioning into monitor mode. However, the SPARC CPU can invoke synchronous register window overflow or underflow exceptions at almost any function call or return. As a result, the tag exception handler must install its own traditional CPU exception handlers before proceeding to execute C code in monitor mode. Since the traditional CPU exception mechanism does not transition the processor either in or out of monitor mode on its own, the security monitor's traditional exception handlers need not be any different than their non-monitor-mode counterparts.

3.7 Devices

One limitation of our prototype is that most device handling code is part of the trusted security monitor. Moreover, because the traditional interrupt mechanism does not switch the processor into monitor mode, device liveness relies on individual kernel instances handing off device interrupts to the driver in the security monitor. (However, if untrusted kernel code cannot clear the interrupt condition, the interrupt will be serviced as soon as the CPU starts executing a well-behaved kernel instance.) We believe that there are two approaches to reducing the amount of trusted device driver code, corresponding to two kinds of devices, as follows.

For devices that only handle data with a single label, such as a network card, a mechanism for controlling both device DMA and access to device registers would be sufficient for moving the device driver into a separate protection domain outside the fully-trusted monitor. The DMA control mechanism could use either memory tagging to define the set of tags accessible to each device, or an IOMMU mechanism like Intel's VT-d [1], although properly implementing protection through translation requires avoiding peer-to-peer bus transactions and other potential pitfalls [24].

The more difficult case is devices that handle differently-labeled data, such as the disk. While the disk device driver would likely remain in trusted code, we expect that support for lightweight tagging of on-disk data would allow moving some of the file system implementation into untrusted code. For example, a small amount of flash available in hybrid disk drives today could be used to store sector-level tag values, and track the label of data as it moves between RAM and disk.

4 MICROARCHITECTURE

Loki enables building secure systems by providing fine-grained, software-controlled permission checks and tag exceptions. This section discusses several key aspects of the Loki design and microarchitecture. Figure 4 shows the overall structure of the Loki pipeline.

4.1 Memory tagging

Loki provides memory tagging support by logically associating an opaque 32-bit tag with every 32-bit word of physical memory. Associating tags with physical memory, as opposed to virtual addresses, avoids potential aliasing and translation issues in the security monitor. These tags are cacheable, similar to data, and have identical locality.

Naively associating a 32-bit tag value with each 32-bit physical memory location would not only double the amount of physical memory, but also impact runtime performance. Setting tag values for large ranges of memory would be prohibitively expensive if it required manually updating a separate tag for each word of memory. Since tags tend to exhibit high spatial locality [29], our design adopts a *multi-granular* tag storage approach in which page-level tags are stored in a linear array in physical memory, called the *page-tag array*, allocated by the monitor code. This array is indexed by the physical page number to obtain the 32-bit tag for that page. These tags are cached in a structure similar to a TLB for performance. Note that this is different from previous work where page-level tags are stored in the TLBs and page tables [29]. Since we do not make any assumptions about the correctness of the MMU code, we must maintain our tags in a separate structure. The monitor can specify fine-grained tags for a page of memory on demand, by allocating a shadow memory page to hold a 32-bit tag for every 32-bit word of data in the original page, and putting the physical address of the shadow page in the appropriate entry in the linear array, along with a bit to indicate an indirect entry. The benefit of this approach is that DRAM need not be modified to store tags, and the tag storage overhead is proportional to the use of fine-grained tags.

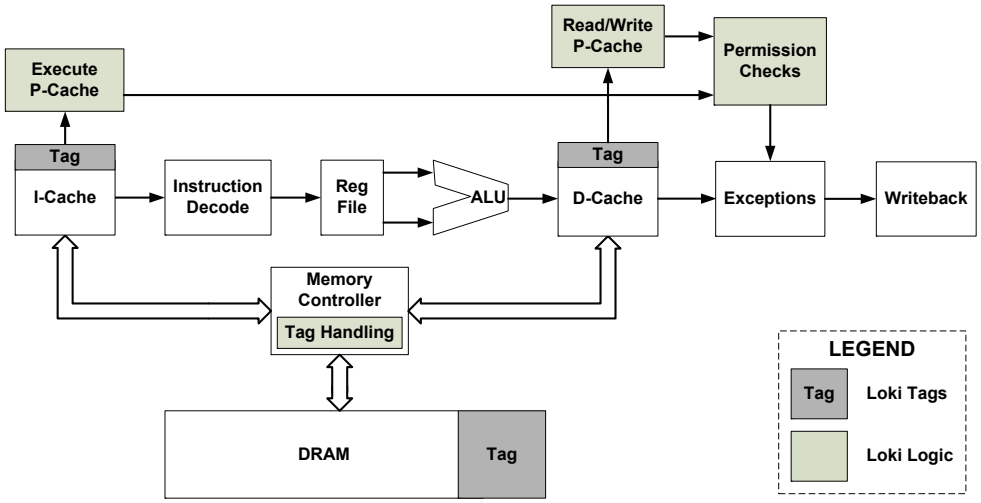


Figure 4: The Loki pipeline, based on a traditional pipelined SPARC processor.

4.2 Permissions cache

Fine-grained permission checks are enforced in hardware using a *permission cache*, or P-cache. The P-cache stores a set of tag values, along with a 3-bit vector of permissions (read, write, and execute) for each of those tag values, which represent the privileges of the currently executing code. Each memory access (load, store, or instruction fetch) checks that the accessed memory location's tag value is present in the P-cache and that the appropriate permission bit is set.

The P-cache is indexed by the least significant bits of the tag. A P-cache entry stores the upper bits of the tag and its 3-bit permission vector. The monitor handles P-cache misses by filling it in as required, similar in spirit to a software-managed TLB. All known TLB optimization techniques apply to the P-cache design as well, such as multi-level caches, separate caches for instruction and data accesses, hardware assisted fills, and so on.

The size of the P-cache, and the width of the tags used, are two important hardware parameters in the Loki architecture that impact the design and performance of software. The size of the P-cache affects system performance, and effectively limits the working set size of application and kernel code in terms of how many different tags are being accessed at the same time. Applications that access more tags than the P-cache can hold will incur frequent exceptions invoking the monitor code to refill the P-cache. However, the total number of security policies specified in hardware is not limited by the size of the P-cache, but by the width of the tag. In our experience, 32-bit tags provide both a sufficient number of tag values, and sufficient flexibility in the design of the tag value representation scheme. Finally, as we will show later in the evaluation of our prototype, even a small number of P-cache entries is sufficient to achieve good performance for a wide variety of workloads.

4.3 Tag exceptions

When a tag permission check fails, control must be transferred to the security monitor, which will either update the permission cache based on the tag of the accessed memory location, or terminate the offending protection domain. Ideally, the exception mechanism will be such that the trusted security handler can be as simple as possible, to minimize TCB size. Traditional trap and interrupt handling facilities do not conform with this, as they rely on the integrity of the MMU state, such as page tables, and privileged registers that may be modified by potentially malicious kernel code.

To address this limitation, Loki introduces a tag exception mechanism that is independent of the traditional CPU exception mechanism. On a tag exception, Loki saves exception information to a few dedicated hardware registers, disables the MMU, switches to the monitor privilege level, and jumps to the tag exception handler in the trusted monitor. The MMU must be disabled because untrusted kernel code has full control over MMU registers and page tables. For simplicity, Loki also disables external device interrupts when handling a tag exception. The predefined address for the monitor is available in a special register introduced by Loki, which can only be updated while in monitor mode, to preclude malicious code from hijacking monitor mode. As all code in the monitor is trusted, tag permission checks are disabled in monitor mode. The monitor also has direct access to a set of registers that contain information about the tag exception, such as the faulting tag.

5 PROTOTYPE EVALUATION

The main goal of this paper was to show that tagged memory support can significantly reduce the amount of trusted code in a system. To that end, this section reports on our prototype implementation of Loki hardware and the complexity and security of our LoStar software prototype. We then show that our prototype performs acceptably by evaluating its performance, and justify our hardware parameter choices by measuring the patterns and locality of tag usage.

In modifying HiStar to take advantage of Loki, we added approximately 1,300 lines of C and assembly code to the kernel, and modified another 300 lines of C code, but the resulting TCB is reduced by 6,400 lines of code—*more than a factor of two*. While Loki greatly reduces the amount of trusted code, we have no formal proof of the system's security. Instead, our current prototype relies manual inspection of both its design and implementation to minimize the risk of a vulnerability.

5.1 Loki prototype

To evaluate our design of Loki, we developed a prototype system based on the SPARC architecture. Our prototype is based on the Leon SPARC V8 processor, a 32-bit open-source synthesizable core developed by Gaisler Research [11]. We modified the pipeline to perform our security operations, and mapped the design to an FPGA board, resulting in a fully functional SPARC system that runs HiStar. This gives us the ability to run real-world applications and gauge the effectiveness of our security primitives.

Leon uses a single-issue, 7-stage pipeline. We modified its RTL code to add support for coarse and fine-grained tags, added the P-cache, introduced the security registers defined by Loki, and added the instructions that manipulate special registers and provide direct access to tags in the monitor mode. We added 6 instructions to the SPARC ISA to read/write memory tags, read/write security registers, write to the permission cache, and return from a tag exception. We also added 7 security registers that store the exception PC, exception nPC, cause of exception, tag of the faulting memory location, monitor mode flag, address of the tag exception handler in the monitor, and the address of the base of the page-tag array. Figure 4 shows the prototype we built.

We built a permission cache using the design discussed in Section 4.2. This cache has 32 entries and is 2-way set associative. During instruction fetch, the tag of the instruction's memory word is read in along with the instruction from the I-cache. This tag is used to check the Execute permission bit. Memory operations—loads and stores—index this cache a second time, using the memory word's tag. This is used to check the Read and Write permission bits. As a result, the permission cache is accessed at least once by every instruction, and twice by some instructions. This requires either two ports into the cache or separate execute and read/write P-caches to allow for simultaneous lookups. Figure 4 shows a simplified version of this design for clarity.

Parameter	Specification
Pipeline depth	7 stages
Register windows	8
Instruction cache	16 KB, 2-way set associative
Data cache	32 KB, 2-way set associative
Instruction TLB	8 entries, fully-associative
Data TLB	8 entries, fully-associative
Memory bus width	64 bits
Prototype Board	Xilinx University Program (XUP)
FPGA device	XC2VP30
Memory	512 MB SDRAM DIMM
Network I/O	100 Mbps Ethernet MAC
Clock frequency	65 MHz

Figure 5: The architectural and design parameters for our prototype of the Loki architecture.

Component	Block RAMs	4-input LUTs
Base Leon	43	14,502
Loki Logic	2	2,756
Loki Total	45	17,258
Increase over base	**5%**	**19%**

Figure 6: Complexity of our prototype FPGA implementation of Loki in terms of FPGA block RAMs and 4-input LUTs.

As mentioned in Section 4.1, we implement a multi-granular tag scheme with a page-tag array that stores the page-level tags for all the pages in the system. These tags are cached for performance in an 8-entry cache that resembles a TLB. Fine-grained tags can be allocated on demand at word granularity. We reserve a portion of main memory for storing these tags and modified the memory controller to properly access both data and tags on cached and uncached requests. We also modified the instruction and data caches to accommodate these tag bits.

We synthesized our design on the Xilinx University Program (XUP) board which contains a Xilinx XC2VP30 FPGA. Figure 5 summarizes the basic board and design statistics, and Figure 6 quantifies the changes made for the Loki prototype by detailing the utilization of FPGA resources. Note that the area overhead of Loki's logic will be lower in modern superscalar designs that are significantly more complex than the Leon. Since Leon uses a write-through, no-write-allocate data cache, we had to modify its design to perform a read-modify-write access on the tag bits in the case of a write miss. This change and its small impact on application performance would not have been necessary with a write-back cache. There was no other impact on the processor performance, as the permission table accesses and tag processing occur in parallel and are independent from data processing in all pipeline stages.

Lines of code	HiStar	LoStar
Kernel code	11,600 (trusted)	12,700 (untrusted)
Bootstrapping code	1,300	1,300
Security monitor code	N/A	5,200 (trusted)
TCB size: *trusted code*	**11,600**	**5,200**

Figure 7: Complexity of the original *trusted* HiStar kernel, the *untrusted* LoStar kernel, and the *trusted* LoStar security monitor. The size of the LoStar kernel includes the security monitor, since the kernel uses some common code shared with the security monitor. The bootstrapping code, used during boot to initialize the kernel and the security monitor, is not counted as part of the TCB because it is not part of the attack surface in our threat model.

5.2 Trusted code base

To evaluate how well the Loki architecture allows an operating system to reduce the amount of trusted code, we compare the sizes of the original, fully trusted HiStar kernel for the Leon SPARC system, and the modified LoStar kernel that includes a security monitor, in Figure 7. To approximate the size and complexity of the trusted code base, we report the total number of lines of code. The kernel and the monitor are largely written in C, although each of them also uses a few hundred lines of assembly for handling hardware traps. LoStar reduces the amount of trusted code in comparison with HiStar by more than a factor of two. The code that LoStar removed from the TCB is evenly split between three main categories: the system call interface, page table handling, and resource management (the security monitor tags pages of memory but does not directly manage them).

5.3 Performance

To understand the performance characteristics of our design, we compare the relative performance of a set of applications running on unmodified HiStar on a Leon processor and on our modified LoStar system on a Leon processor with Loki support. The application binaries are the same in both cases, since the kernel interface remains the same. We also measure the performance of LoStar while using only word-granularity tags, to illustrate the need for page-level tag support in hardware.

Figure 8 shows the performance of a number of benchmarks. Overall, most benchmarks achieve similar performance under HiStar and LoStar (overhead for LoStar ranges from 0% to 4%), but support for page-level tags is critical for good performance, due to the extensive use of page-level memory tagging. For example, the page allocator must change the tag values for all of the words in an entire page of memory in order to give a particular protection domain access to a newly-allocated page. Conversely, to revoke access to a page from a protection domain when the page is freed, the page allocator must reset all tag values back to a special tag value that no other protection domain can access. Explicitly setting tags for each of the words in a page incurs a significant performance penalty (up to 55%), and being able to adjust the tag of a page with a single memory write greatly improves performance.

Compute-intensive applications, represented by the *primes* and *gzip* workloads, achieve the same performance in both cases (0% overhead). Even system-intensive applications that do not switch protection domains, such as the system call and file system benchmarks, incur negligible overhead (0-2%), since they rarely invoke the security monitor. Applications that frequently switch between protection domains incur a slightly higher overhead, because all protection domain context switches must be done through the security monitor, as illustrated by the *IPC ping-pong* workload (2% overhead). However, LoStar achieves good network I/O performance, despite a user-level TCP/IP stack that causes significant context switching, as can be seen in the *wget* workload (4% overhead). Finally, creation of a new protection domain, illustrated by the *fork/exec* workload, involves re-labeling a large number of pages, as can be seen from the high performance overhead (55%) without page-level tags. However, the use of page-level tags reduces that overhead down to just 1%.

5.4 Tag usage and storage

To evaluate our hardware design parameters, we measured the tag usage patterns of the different workloads. In particular, we wanted to determine the number of pages that require fine-grained word-level tags versus the number of pages where all of the words in the page have the same tag value, and the working set size of tags—that is, how many different tags are used at once by different workloads. Figure 9 summarizes our results for the workloads from the previous sub-section.

The results show that all of the different workloads under consideration make moderate use of fine-grained tags. The primary use of fine-grained tags comes from protecting the metadata of each kernel object. For example, workloads with a large number of small files, each of which corresponds to a separate kernel object, require significantly more pages with fine-grained tags compared to a workload that uses a small number of large files. Since Loki implements fine-grained tagging for a page by allocating a shadow page to store a 32-bit tag for each 32-bit word of the original page, tag storage overhead for such pages is 100%. On the other hand, pages storing user data (which includes file contents) have page-level tags, which incur a much lower tag storage overhead of $4/4096 \approx 0.1\%$. As a result, overall tag storage overhead is largely influenced by the average size of kernel objects cached in memory for a given workload. We expect that it's possible to further reduce tag storage overhead for fine-grained tags by using a more compact in-memory representation, like the one used by Mondriaan Memory Protection [33], or to rearrange ker-

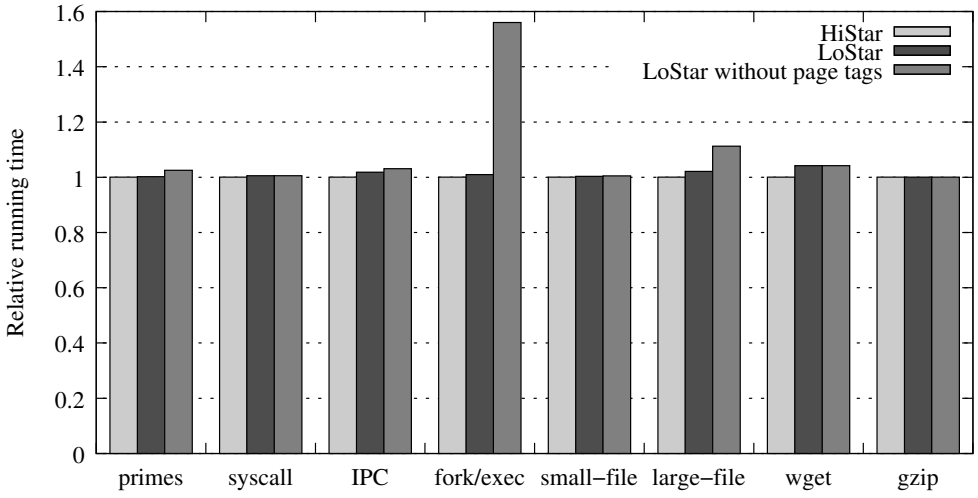


Figure 8: Relative running time (wall clock time) of benchmarks running on unmodified HiStar, on LoStar, and on a version of LoStar without page-level tag support, normalized to the running time on HiStar. The *primes* workload computes the prime numbers from 1 to 100,000. The *syscall* workload executes a system call that gets the ID of the current thread. The *IPC ping-pong* workload sends a short message back and forth between two processes over a pipe. The *fork/exec* workload spawns a new process using `fork` and `exec`. The *small-file* workload creates, reads, and deletes 1000 512-byte files. The *large-file* workload performs random 4KB reads and writes within a single 4MB file. The *wget* workload measures the time to download a large file from a web server over the local area network. Finally, the *gzip* workload compresses a 1MB binary file.

Workload	primes	syscall	IPC	fork/exec	small files	large files	wget	gzip
Fraction of memory pages with word-granularity tags	40%	49%	54%	65%	58%	3%	18%	16%
Maximum number of concurrently accessed tag values	12	11	18	24	13	13	30	12

Figure 9: Tag usage under different workloads running on LoStar.

nel data structures to keep data with similar tags together, although doing so would likely increase complexity either in hardware or software.

Finally, all workloads shown in Figure 9 exhibit reasonable tag locality, requiring only a small number of tags at time. This supports our design decision to use a small fixed-size hardware permission cache.

6 Related work

Since this paper describes a combination of hardware and software, we will discuss related work in these two areas in turn.

6.1 Hardware

Many hardware protection architectures have been previously proposed. Multics [25] introduced hierarchical protection rings which were used to isolate trusted code in a coarse-grained manner. *x*86 processors also have 4 privilege levels, but the page table mechanism can only distinguish between two effective levels. However, application security policies are often not hierarchical, and Loki's 32-bit tag space provides a way of representing a large number of such policies in hardware.

The Intel i432 and Cambridge CAP systems, among others [20], augment the way applications name memory with a capability, which allows enforcing non-hierarchical security policies by controlling access to capabilities, at the cost of changing the way software uses pointers. Loki associates security policies with physical memory, instead of introducing a name translation mechanism to perform security checks. As a result, the security policy for any piece of data in Loki is always unambiguously defined, regardless of any aliasing that may be present in higher-level translation mechanisms.

The protection lookaside buffer (PLB) [16] provides a similarly non-hierarchical access control mechanism for a global address space (although only at page-level granularity). While the PLB caches permissions for virtual addresses, Loki's permissions cache stores permissions in terms of tag values, which is much more compact, as Section 5.4 suggests.

The IBM system i [13] associates a one-bit tag with physical memory to indicate whether the value represents a pointer or not. Similarly, the Intel i960 [14] provides a one-bit tag to protect kernel memory. Loki's tagged memory architecture is more general, providing a large number of protection domains.

Mondriaan Memory Protection (MMP) [33] provides lightweight, fine-grained (down to individual memory words) protection domains for isolating buggy code. However, MMP was not designed to reduce the amount

of trusted code in a system. Since the MMP supervisor relies on the integrity of the MMU and page tables, MMP cannot enforce security guarantees once the kernel is compromised. Loki extends the idea of lightweight protection domains to physical resources, such as physical memory, to achieve benefits similar to MMP's protection domains with stronger guarantees and a much smaller TCB. Moreover, this paper describes how a fine-grained memory protection mechanism can be used to extend the enforcement of application security policies all the way down into hardware.

The Loki design was initially inspired by the Raksha hardware architecture [9]. However, the two systems have significant design differences. Raksha maintains four independent one-bit tag values (corresponding to four security policies) for each CPU register and each word in physical memory, and propagates tag values according to customizable tag propagation rules. Loki, on the other hand, maintains a single 32-bit tag value for each word of physical memory (allowing the security monitor to define how multiple security policies interact), does not tag CPU registers, and does not propagate tag values. Raksha's propagation of tag values was necessary for fine-grained taint tracking in unmodified applications, but it could not enforce write-protection of physical memory. Conversely, Loki's explicit specification of tag values works well for a system like HiStar, where all state in the system already has a well-defined security label that controls both read and write access.

There has also been significant work on hardware support for other types of security mechanisms, such as dynamic information flow tracking, to prevent attacks such as buffer overflows [6, 8, 9, 29]. Hardware designs for preventing information leaks in user applications have also been proposed [28, 32], although these designs do not attempt to reduce the TCB size. None of these designs provide a sufficiently large number of protection domains needed to capture different application security policies. Moreover, enforcement of information flow control in hardware has inherent covert channels relating to the re-labeling of physical memory locations. HiStar's system call interface avoids this by providing a virtually unlimited space of kernel object IDs that are never relabeled.

6.2 Software

Many operating systems, including KeyKOS [4], EROS [27], and HiStar [35], provide strong isolation of application code using a small, fully trusted kernel. However, existing hardware architectures fundamentally require that the fully trusted kernel include code to manage page tables, device drivers, and so on, in order to provide different protection domains for user-level code. LoStar can enforce certain security guarantees using a significantly smaller trusted code base, by directly specifying security policies for physical resources in hardware. This allows the fully trusted code base to exclude complex code such as page table management and device drivers. Even for an operating system such as HiStar, where the kernel is already small, Loki allows significantly reducing the trusted code size.

A number of systems attempt to provide some guarantees even in the case of buggy or malicious kernel code. Separation kernels [23] and virtual machine monitors [15] provide strong isolation between multiple processes on a single machine. SecVisor [26] ensures kernel code integrity in a small hypervisor. Proxos [31] allows sensitive applications to partition trust in operating system abstractions by using an untrusted kernel for certain peripheral functionality. Flicker [21] enables tamper-proof code execution without trusting the underlying operating system. Nooks [30] and Mondrix [34] isolate potentially buggy device driver code in the Linux kernel. These systems enforce relatively static security policies that do not directly map onto application security goals. As a result, applications running on top of these systems must provide their own security enforcement mechanisms. In contrast, LoStar maps application security policies onto the underlying hardware protection mechanisms, providing strong enforcement of application security.

Singularity [12] avoids the need for hardware protection mechanisms by relying on type safety instead. However, we believe that Singularity could also benefit from associating security policies with data, perhaps using types.

The VMM security kernel [15] provides strong isolation across multiple virtual machines with limited sharing. Using the Loki architecture, the VMM security kernel could be implemented using significantly less trusted code, by directly specifying security policies for physical hardware resources used by the different virtual machines. Unlike LoStar, the VMM security kernel provides very limited sharing. A virtual machine monitor could adopt an interface similar to that provided by Loki to enforce security policies on behalf of applications running inside a virtual machine.

Overshadow [7] aims to protect application data in an untrusted OS by using a virtual machine monitor. One of the most complex aspects of Overshadow is providing a secure binding between application names (such as Unix pathnames) and protection domains. LoStar addresses this problem by relying on HiStar's design which reduces all naming to a single 61-bit kernel object namespace. As a result, LoStar needs only to ensure the integrity of a single flat namespace in the trusted security monitor, which is simpler than a hierarchical file system.

7 Conclusion

This paper showed how hardware support for tagged memory can be used to enforce application security policies. We presented Loki, a hardware tagged memory architecture that provides fine-grained, software-managed access control for physical memory. We also showed how HiStar, an existing operating system, can take advantage of Loki by directly mapping application security policies to the hardware protection mechanism. This allows the amount of trusted code in the HiStar kernel to be reduced by over a factor of two. We built a full-system prototype of Loki by modifying a synthesizable SPARC core, mapping it to an FPGA board, and porting HiStar to run on it. The prototype demonstrates that our design can provide strong security guarantees while achieving good performance for a variety of workloads in a familiar Unix environment.

Acknowledgments

We thank Silas Boyd-Wickizer for porting HiStar to the SPARC processor. We also thank Bryan Ford, David Mazières, Michael Walfish, the anonymous reviewers, and our shepherd, Galen Hunt, for their feedback. This work was funded by NSF Cybertrust award CNS-0716806, by NSF award CCF-0701607, by joint NSF Cybertrust and DARPA grant CNS-0430425, by NSF through the TRUST Science and Technology Center, and by Stanford Graduate Fellowships supported by Cisco Systems and Sequoia Capital.

References

[1] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Weigert. Intel Virtualization Technology for directed I/O. *Intel Technology Journal*, 10(3):179–192, August 2006.

[2] D. E. Bell and L. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corp., Bedford, MA, March 1976.

[3] K. J. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, MITRE Corp., Bedford, MA, April 1977.

[4] A. C. Bomberger, A. P. Frantz, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *Proc. of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, April 1992.

[5] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *Proc. of the 11th USENIX Security Symposium*, San Francisco, CA, August 2002.

[6] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Proc. of the International Conference on Dependable Systems and Networks*, Yokohama, Japan, June 2005.

[7] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. of the 13th ASPLOS*, Seattle, WA, March 2008.

[8] J. R. Crandall and F. T. Chong. MINOS: Control data attack prevention orthogonal to memory model. In *Proc. of the 37th Intl. Symposium on Microarchitecture*, Portland, OR, December 2004.

[9] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proc. of the 34th ISCA*, San Diego, CA, June 2007.

[10] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *Proc. of the 20th SOSP*, pages 17–30, Brighton, UK, October 2005.

[11] Gaisler Research. LEON3 SPARC Processor. `http://www.gaisler.com`.

[12] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft, Redmond, WA, October 2005.

[13] IBM Corporation. IBM system i. `http://www-03.ibm.com/systems/i`.

[14] Intel Corporation. Intel i960 processors. `http://developer.intel.com/design/i960/`.

[15] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A VMM security kernel for the VAX architecture. In *Proc. of the 1990 IEEE Symposium on Security and Privacy*, pages 2–19, Oakland, CA, May 1990.

[16] E. Koldinger, J. Chase, and S. Eggers. Architectural support for single address space operating systems. Technical Report 92-03-10, University of Washington, Department of Computer Science and Engineering, March 1992.

[17] M. Krohn. Building secure high-performance web services with OKWS. In *Proc. of the 2004 USENIX*, June–July 2004.

[18] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proc. of the 21st SOSP*, Stevenson, WA, October 2007.

[19] B. Lampson, M. Abadi, M. Burrows, and E. P. Wobber. Authentication in distributed systems: Theory and practice. *ACM TOCS*, 10(4):265–310, 1992.

[20] H. M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[21] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. How low can you go? Recommendations for hardware-supported minimal TCB code execution. In *Proc. of the 13th ASPLOS*, Seattle, WA, March 2008.

[22] President's Information Technology Advisory Committee (PITAC). Cybersecurity: A crisis of prioritization. `http://www.nitrd.gov/pitac/reports/20050301_cybersecurity/cybersecurity.pdf`, February 2005.

[23] J. M. Rushby. Design and verification of secure systems. In *Proc. of the 8th SOSP*, pages 12–21, Pacific Grove, CA, December 1981.

[24] J. Rutkowska and R. Wojtczuk. Preventing and detecting Xen hypervisor subversions. `http://invisiblethingslab.com/bh08/part2-full.pdf`, August 2008.

[25] M. D. Schroeder and J. H. Saltzer. A hardware architecture for implementing protection rings. *Comm. of the ACM*, 15(3):157–170, 1972.

[26] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proc. of the 21st SOSP*, pages 335–350, October 2007.

[27] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Proc. of the 17th SOSP*, December 1999.

[28] W. Shi, H.-H. Lee, G. Gu, L. Falk, T. Mudge, and M. Ghosh. InfoShield: A security architecture for protecting information usage in memory. In *Proc. of the 12th HPCA*, Austin, TX, February 2006.

[29] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proc. of the 11th ASPLOS*, Boston, MA, October 2004.

[30] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM TOCS*, 23(1), 2005.

[31] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proc. of the 7th OSDI*, pages 279–292, Seattle, WA, November 2006.

[32] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proc. of the 37th International Symposium on Microarchitecture*, Portland, OR, December 2004.

[33] E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. In *Proc. of the 10th ASPLOS*, San Jose, CA, October 2002.

[34] E. Witchel, J. Rhee, and K. Asanovic. Mondrix: Memory isolation for linux using mondriaan memory protection. In *Proc. of the 20th SOSP*, pages 31–44, October 2005.

[35] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th OSDI*, pages 263–278, Seattle, WA, November 2006.

[36] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *Proc. of the 5th NSDI*, pages 293–308, San Francisco, CA, April 2008.

Device Driver Safety Through a Reference Validation Mechanism *

Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, Fred B. Schneider
Cornell University

Abstract

Device drivers typically execute in supervisor mode and thus must be fully trusted. This paper describes how to move them out of the trusted computing base, by running them without supervisor privileges and constraining their interactions with hardware devices. An implementation of this approach in the Nexus operating system executes drivers in user space, leveraging hardware isolation and checking their behavior against a safety specification. These Nexus drivers have performance comparable to in-kernel, trusted drivers, with a level of CPU overhead acceptable for most applications. For example, the monitored driver for an Intel e1000 Ethernet card has throughput comparable to a trusted driver for the same hardware under Linux. And a monitored driver for the Intel i810 sound card provides continuous playback. Drivers for a disk and a USB mouse have also been moved successfully to operate in user space with safety specifications.

1 Introduction

Device drivers constitute over half of the source code of many operating system kernels, with a bug rate up to seven times higher than other kernel code [10]. They are often written by outside developers, and they are less rigorously examined and tested than the rest of the kernel code. Yet device drivers are part of the trusted computing base (TCB) of every application, because the monolithic architecture of mainstream operating systems forces device drivers to be executed inside the kernel, with high privilege. Some microkernels and other research operating systems [2, 9, 21, 24] run device drivers in user space to isolate the operating system from accidental driver faults, but these drivers retain sufficient I/O privileges that they must still be trusted.

This paper introduces a practical mechanism for executing device drivers in user space and without privilege. Specifically, device drivers are isolated using hardware protection boundaries. Each device driver is given access only to the minimum resources and operations necessary to support the devices it controls (least privilege), thereby shrinking the TCB.[1] A system in which device drivers have minimal privileges is easier to audit and less susceptible to Trojans in third-party device drivers.

Even in user space, device drivers execute hardware I/O operations and handle interrupts. These operations can cause device behavior that compromises the integrity or availability of the kernel or other programs. Therefore, our driver architecture introduces a global, trusted *reference validation mechanism* (RVM) [3] that mediates all interaction between device drivers and devices. The RVM invokes a device-specific *reference monitor* to validate interactions between a driver and its associated device, thereby ensuring the driver conforms to a *device safety specification* (DSS), which defines allowed and, by extension, prohibited behaviors.

The DSS is expressed in a domain-specific language and defines a state machine that accepts permissible transitions by a monitored device driver. We provide a compiler to translate a DSS into a reference monitor that implements the state machine. Every operation by the device driver is vetted by the reference monitor, and operations that would cause an illegal transition are blocked. The entire architecture is depicted in Figure 1.

The RVM protects the integrity, confidentiality, and availability of the system, by preventing:

- **Illegal reads and writes:** Drivers cannot read or modify memory they do not own.

*Supported by NICECAP cooperative agreement FA8750-07-2-0037 administered by AFRL, AFOSR grant F49620-03-1-0156, National Science Foundation Grants 0430161 and CCF-0424422 (TRUST), ONR Grant N00014-01-1-0968, and Microsoft Corporation.

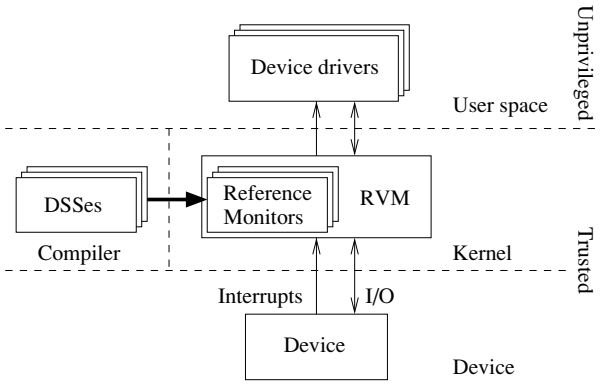


Figure 1: Safe user-space device driver architecture.

- **Priority escalation:** Drivers cannot escalate their scheduling priority.
- **Processor starvation:** Drivers cannot hold the CPU for more than a pre-specified number of time slices.
- **Device-specific attacks:** Drivers cannot exhaust device resources or cause physical damage to devices.

In addition, given a suitable DSS, an RVM can enforce site-specific policies to govern how devices are used. For example, administrators at confidentiality-sensitive organizations might wish to disallow the use of attached microphones or cameras; or administrators of trusted networks might wish to disallow promiscuous (sniffing) mode on network cards.

One alternative to our approach for monitoring and constraining device driver behavior is to use hardware capable of blocking illegal operations. Hardware-based approaches, however, are necessarily limited to policies expressed in terms of hardware events and abstractions. An IOMMU [1, 4, 14, 23], for example, can limit the ability of devices to perform DMA transfers to or from physical addresses the associated drivers cannot read or write directly. IOMMUs, however, do not mediate aspects of driver and system safety that go beyond the memory access interface [7]; for example, an IOMMU cannot prevent interrupt livelock, limit excessively long interrupt processing, protect devices from physical harm by drivers, or enforce site-specific policies. As IOMMUs become prevalent, our approach could leverage them as hardware accelerators for memory protection.

In sum, this paper shows how to augment common memory protection techniques with device-specific reference monitors to execute drivers with limited privilege and in user space. The requisite infrastructure is small, easy to audit, and shared across all devices. Our prototype implementation demonstrates that this approach can defend against malicious drivers and that the performance costs of this enhanced security are not prohibitive.

2 Device I/O Model

Device drivers send commands to devices, check device status using registers, receive notification of status changes through interrupts, and initiate bulk data transfers using direct memory access (DMA). How they do so constitutes a platform's *I/O model*. Our work is targeted to the x86 architecture and PCI buses; what follows is a brief overview of the I/O model on that platform. Similar features are found on other processors and buses.

Modern buses implement device enumeration and endpoint identification. Each device on a PCI bus is identified by a 16-bit vendor identifier and a 16-bit model number; the resulting 32-bit *device identifier* identifies the device.[2] Some devices with different model numbers may nonetheless be similar enough to share a single driver and a single DSS. *Device enumeration* is a process for identifying all devices attached to a bus; *endpoint identification* is the process of querying a device for its type, capabilities, and resource requirements.

Device enumeration and endpoint identification typically occur at boot time. Interrupt lines and I/O registers are assigned, according to device requests, to all devices discovered. Device identifiers govern which device drivers to load. Unrecognized devices, for which no DSS is available, are ignored and are not available to drivers.

Devices have *registers*, which are read and written by drivers to get status, send commands, and transfer data. The registers comprise I/O ports (accessed using instructions like `inb` and `outb`), memory-mapped I/O, and PCI-configuration registers. Each register is identified by a *type* and an *address*. Contiguous sets of registers constitute a *range*, identified by type, base address, and limit (the number of addresses in the range). For all register types, accesses are parameterized by an address, a size, and, for writes, a value of the given size. Write operations elicit no response; read operations produce a value of the given size as a response. Both operations can cause side effects on a device.

Devices that transfer large amounts of data typically employ DMA rather than requiring a device driver to transfer each word of data individually through device registers. Before initiating a DMA transfer, the device driver typically sets a control register on the device to point to a buffer in memory. Some devices can perform DMA to or from multiple memory locations; in this case, a control register might contain a pointer to a list, ring,

or tree structure with pointers to many buffers. Device drivers using DMA transfers must first obtain from the kernel one or more memory regions with known, fixed, physical addresses.

Devices can be *synchronous* or *asynchronous*. Drivers must poll synchronous devices for completed operations or changes in status. In contrast, when a driver submits an operation to an asynchronous device, the driver yields the CPU until the device later signals its response (or other status change) by interrupting the processor. When that interrupt occurs, the operating system invokes code specified by the driver. In most cases, an interrupt must be acknowledged by a driver, or the device will continue to send the same interrupt. Interrupts can be prioritized relative to each other, but they generally occur with a high priority, preempting most other tasks.

Each device signals interrupts using a pre-assigned *interrupt line*. On some architectures, including the x86, interrupt lines can be shared by multiple devices. Drivers must read status registers for each of these devices to determine which specific device caused the interrupt.

Devices are assumed to be in an unknown state when an operating system boots or when a driver is loaded or reloaded. When a driver is unloaded, it unregisters its interrupt handler and releases its DMA memory. At that point, the device must be placed in a state that does not generate interrupts or use DMA.

Devices are typically forgiving about device driver timing, and device drivers are similarly forgiving about device timing. This flexibility is a necessity, because a modern multitasking operating system might be heavily loaded, implement arbitrary scheduling policies, or at times execute with interrupts disabled. In addition, devices and their drivers typically work with several processor generations, which differ in execution speed. Device registers and interrupts, rather than precise timing, are used to implement synchronization between the device and its driver so that devices and drivers behave safely and predictably despite uncertain delays.

Some drivers are divided into components or hierarchies. For example, SCSI, ATA, and USB each have a controller driver plus additional drivers for peripherals, like disks, mice, keyboards, etc. In the general case, any driver in such a *driver hierarchy* can issue requests and handle interrupts. Applying a reference monitor in such a driver hierarchy requires the reference monitor to securely identify attached devices, demultiplex the commands passing through the controller, and recognize the protocol used with each device—all feasible with our current language.[3] However, in all driver hierarchies we have examined, only the device driver for the controller performs low-level I/O operations, handles interrupts, or initiates DMA transfers, and drivers for peripherals communicate with their devices through the controller driver. Hence, all communication is visible to a single reference monitor, which suffices to validate the operations of all drivers in the hierarchy.

Some devices, particularly high-performance network cards and 3-D graphics cards, support loading and executing programs (e.g., for TCP offload or vertex shading) on a processor on the device. Other devices may support loading firmware, either ephemerally or permanently. Such programs and firmware change the way the device behaves; thus, they must be trustworthy. Programs and firmware are loaded through I/O operations or DMA, both of which can be monitored. In principle, then, an RVM could authenticate device programs or firmware using signatures or other analysis. Our current DSSes do not implement these checks. Doing so would be straightforward, though designing an analysis algorithm for such updates might not be.

3 Unprivileged Driver Architecture

In our user-space driver architecture, drivers, like any other user process, are loaded from a filesystem; once loaded, they execute and can be unloaded and restarted at any time. When a driver is first loaded, it executes a system call to find a compatible device. As part of this system call execution, the RVM identifies an appropriate device and reference monitor and returns to the driver a structure describing the device ID and I/O-resource assignments. The driver then uses driver system calls (described in Section 4.3) to perform I/O operations and receive interrupts. Subsequent uses of those calls cause the RVM to invoke the reference monitor.

Reference monitors are instantiated immediately after endpoint enumeration, based on device IDs. Reference monitors persist, even if corresponding drivers are unloaded and restarted.

3.1 Security properties

Drivers are not trusted, but the RVM, reference monitors, and devices are. Moreover, reference monitors are compiled from DSSes, so DSSes and the DSS compiler must be trusted.

Some DSSes will be written by hardware manufacturers; others will be written by independent experts, including security firms or OS distributors. But independent of the source, a DSS ought to be small and declarative. Further, because they describe devices, not drivers,

there need only be one DSS per device. Hence, they are conducive to auditing.

We assume devices behave safely if given sufficiently restricted inputs. Such an assumption is inescapable, because devices can access any memory, generate arbitrary interrupts, and starve hardware buses directly.

The two sources of driver misbehavior we consider are drivers designed by malicious authors (Trojans), and drivers with bugs that can be subverted by users or remote attackers. Both are dealt with by our RVM.

The RVM prevents drivers from performing invalid reads and writes using hardware isolation and by checking driver accesses to DMA control registers.

- Hardware isolation works as with other user processes, giving each driver process direct access only to its own memory space.
- By checking that every DMA address sent to the device is allocated to the driver, the RVM prevents a device driver from using DMA for illegal reads and writes.

The RVM must also defend against a device driver that attempts to escalate its execution priority or that starves other processes and the kernel by causing large numbers of interrupts or by spending too much time in high-priority interrupt handlers. A timer driver might set too high a timer frequency, or a sound card driver might set too small a DMA buffer for playback, causing frequent notifications to be generated when the buffer becomes empty. Some of these unacceptable behaviors can be prevented when the driver is setting up the device—for example, by a reference monitor imposing a lower bound on the sound card DMA buffer size—but RVMs provide three additional protection measures. First, the RVM limits the frequency at which a driver can receive interrupts, with different limits for different types of devices. Second, the RVM limits the length of time that an interrupt handler runs. Third, the RVM ensures that each interrupt handler acknowledges every interrupt, to prevent devices from issuing additional interrupts for the same event. (The details of monitoring interrupt handlers in our Nexus implementation are described in Section 4.1.)

Finally, an RVM must prevent invocations of operations known or suspected to harm devices. Examples include: overclocking processors, sending a monitor an out-of-range refresh rate, instructing a disk to seek to an invalid location, or writing invalid data to non-volatile configuration registers. Other attacks against devices involve exhausting finite resources, such as wearing out flash memory with excessive writes or wasting battery power on mobile devices. The RVM prevents many such attacks by allowing only well-defined operations at rates presumed to be safe.

While the RVM approach is general enough to enforce rich safety properties, we do not anticipate that RVMs will be used to enforce driver semantics expected by applications. Our reference monitor implementations do not, for example, ensure that network drivers only send legal TCP packets. They also do not prevent a malicious driver from providing incorrect or incomplete access to a device (i.e. denial of service). Such protections concern end-to-end properties, hence we believe that they are best implemented above the driver level.

3.2 Device safety specifications (DSS)

Each DSS describes the *states* and *transitions* for a *state machine* and is compiled to create a reference monitor. Inputs to the reference monitor—operations executed by a driver and events from the corresponding device—are delivered serially to the reference monitor by the RVM. When an input does not correspond to an allowable transition, then the reference monitor deems it illegal, the RVM terminates the driver for the corresponding device, and the device is reset.

The state of a DSS state machine records interesting aspects of the history of operations and events. This state is defined in terms of *state variables*, and it often correlates with the state of the I/O device itself. Some of these state variables are explicitly defined by the program; others are implicitly defined by the RVM.

Implicitly defined state variables are given values by the RVM as a result of registration events (see Section 4.1). The implicit variables `$PORTIO[]`, `$MMIO[]`, `$PCIREG[]`, and `$INTR[]` identify I/O registers and interrupt lines set during endpoint identification. And `$MONITORED[]` and `$UNMONITORED[]` describe two types of memory regions allocated by the driver, both of which may be used for DMA transfers. Access to a monitored memory location generates an input to the reference monitor, similar to device registers; this form of memory is used to store commands or pointers to other DMA regions. Access to an unmonitored memory location is not visible to the RVM, making unmonitored memory suitable only for DMA buffers containing data irrelevant to the DSS, such as audio samples from a sound card. Unmonitored reads and writes are considerably faster than monitored reads and writes.

Each state machine transition is specified with a predicate P_i and an action A_i. P_i is a boolean expression over events and state variables. A_i is a program fragment that

modifies state variables to produce the new state. A transition that pairs a predicate P_i and an action A_i is written using the syntax P_i { A_i }.[4]

Any operation or event—though this is most useful for interrupts—can be assigned a rate limit as part of a DSS. Rate limits can be manually incorporated into transitions using counters and timers. As a convenience, the notation P_i <$rate$, max, $start$> { A_i } compiles to a transition with a leaky bucket expressing a rate limit. So, the associated transition can occur at most $rate$ times per second; bursts are allowed beyond this rate, up to max occurrences at once; when the driver starts, it has $start$ initial capacity.

As an example, an abridged version of our DSS for the Intel i810 audio device appears in the Appendix.

4 Implementation

We instantiated our user-level device driver architecture in the Nexus trusted operating system [28], which has many similarities to traditional microkernels, including hardware-implemented process isolation. Other operating systems that support process isolation (e.g., Linux or Windows) could also host an RVM.

Our implementation of user-space, unprivileged device drivers in Nexus includes the RVM, an event interface between the RVM and the reference monitor, a system call interface by which drivers can request services from the RVM, and a mechanism for limiting driver execution time and the frequency of events. We discuss each of these below and report on our experience porting Linux kernel device drivers to Nexus user space.

4.1 Reference monitor interface in Nexus

Reference monitors define functions that the RVM calls to initialize implicit state variables and to deliver inputs to be checked. These inputs are sent in response to driver system calls and device events. Each I/O operation and event described in Section 2 causes a distinct input.

State-variable setup. After device enumeration and endpoint identification occur, Nexus initializes one reference monitor for each device. The implicit state variables are arrays. The RVM populates them based on the results of endpoint enumeration by calling the function `register_region` to set up I/O ports, memory-mapped I/O, and PCI configuration registers and the function `register_intr` to set up an interrupt line.

Driver and device events. Device drivers affect the state of the system and the reference monitor in three ways: by performing I/O, by allocating memory, or by exiting. When the driver reads or writes a register or a monitored memory location, the RVM sends `read` or `write` events to the reference monitor. After a `read` operation, the device responds with a value, generating a `read_response` event. The `read` operation can be blocked if it would cause a disallowed side effect. The `read_response` event is never blocked, and the value it conveys can be used to change state variables.

A driver can allocate memory to use for DMA, which causes the RVM to send `register_region` events with a region type of `MONITORED` or `UNMONITORED`. Finally, if the driver exits or executes an operation not permitted by the DSS, the RVM sends a `reset` event.

Devices affect reference monitor state when sending interrupts, which generate `intr` events. When an interrupt occurs, the reference monitor sets an interrupt `status` flag (each reference monitor maintains one such flag per interrupt line) to `pending`, and the RVM schedules the driver with high execution priority. The driver then has a configurable amount of time to respond to the interrupt, by checking if the interrupt was from its device, and if so, acknowledging it so the device does not generate more interrupts for the same device event. This check and acknowledgment are implemented with I/O device `read` and `write` operations; reference monitors recognize them as transitions and reset the interrupt `status` flag to `idle`. Then, the RVM lowers the driver's execution priority to its default level. If the driver does not check and acknowledge the interrupt before the allowed time has elapsed,[5] the RVM infers a starvation attack, terminates the driver, and resets the device.

When an interrupt occurs on a shared line, the RVM notifies all drivers on that line. The RVM monitors the handlers to ensure that each driver checks its device's interrupt status and acknowledges the interrupt if necessary. This approach correctly handles merged interrupts, where two or more devices generate an interrupt at the same time, as well as spurious interrupts.

4.2 Rate limiting in Nexus

A device managed by a well-behaved driver should not exceed rate limits enforced by the reference monitor. Drivers can call `driver_get_rate_limits` to learn such rate limits and can manage interrupts using a throttling mechanism provided by the device or by disabling interrupt-generating acts by the device when an interrupt would be disallowed.

The RVM could impose rate limits on uncooperative drivers directly or by terminating a driver when its associated device exceeds the limit. We implement the latter in Nexus. If an RVM can mask interrupts from each device independently (e.g., as with non-shared interrupts or edge- or message-signaled interrupts), then the RVM could limit the interrupt rate by masking interrupts that would exceed a rate limit. However, for shared, level-triggered interrupt lines, this approach delays interrupts for all drivers sharing the line. Since limits cannot be enforced by masking these interrupts, the driver associated with a device that violates rate limits must be terminated.

To ensure that rate limits are applied fairly to interrupts on shared lines, only acknowledged interrupts are counted. The RVM determines from reference monitor state how each driver handled an interrupt—by deciding it was for a different driver, or by acknowledging it.

4.3 System calls in Nexus

Nexus implements system calls for drivers to find a device, allocate memory, and perform I/O operations:

- `driver_init_pci(pci_ids[], &device)` is the main initialization routine. A device driver calls it to find devices and to find their I/O registers and interrupt lines. The first parameter is a list of PCI IDs the driver can manage. The `device` parameter returns a structure describing the I/O registers and interrupt lines for the driver to communicate with the device.
- `driver_allocate_memory(size, is_monitored, &v_addr, &p_addr)` allocates kernel memory for DMA buffers and returns the virtual and physical addresses to the device driver. The `is_monitored` parameter indicates if reads and writes should be checked by the reference monitor. If the allocated region is unmonitored, then the reference monitor will not allow pointers to that region to be written to registers that require monitored memory, such as DMA indices and command buffers.
- `driver_wait_for_intr(intr)` blocks the calling thread in the device driver until an interrupt arrives on the specified interrupt line. Normally, one thread in a driver runs a loop that executes this system call and runs an interrupt handler when the call returns.
- `driver_get_rate_limits()` returns rate limits for all transitions as an array of leaky bucket definitions. A driver can use this information to delay operations and interrupts so that no behavior exceeds rate limits.
- `driver_read(region, addr, len)` and `driver_write(region, addr, len, val)` read and write port I/O, memory-mapped I/O, PCI configuration registers, and monitored DMA memory.

4.4 Driver source compatibility

Rather than write new drivers for Nexus, we used drivers from Linux 2.4.22.[6] Our original goal was source compatibility between these Linux drivers and Nexus user space drivers. However, the Linux drivers did not provide some of the information necessary to enforce a DSS efficiently. Moreover, small changes to driver source code promised to reduce our overall effort in porting Linux drivers to Nexus and to make the resulting Nexus drivers more efficient. So we used a hybrid approach, implementing general-purpose compatibility functions for Linux drivers and also changing Linux driver code to work better with an RVM. The compatibility functions provide user-space equivalents of global variables and functions in the Linux kernel that Linux drivers would normally access directly.

Linux I/O operations. Linux drivers use functions and macros for most I/O operations. Port I/O and MMIO are implemented by macros for reading and writing each valid word size. PCI register I/O is implemented using functions. For our Nexus port, we redefined these macros and functions to call `driver_read` and `driver_write`.

Linux drivers read and write DMA memory by dereferencing pointers or by calling functions like `memcpy`. We map monitored DMA memory to invalid pages so that accessing it causes page faults. A trap handler redirects these page faults to `driver_read` and `driver_write` system calls. System calls are faster than page faults (see Section 5.1), so programmers may change monitored DMA memory operations to explicit system calls wherever performance is critical.

Linux memory allocation. The Linux kernel provides a variety of memory allocation functions, which we redefine to call `driver_allocate_memory`, which implements the subset of memory allocation functionality needed by our drivers. The `driver_allocate_memory` call provides contiguous memory with known addresses appropriate for DMA. Memory without DMA or concurrency requirements is

allocated from the user-space heap. To provide allocation in an interrupt context without deadlocking, we implemented pre-allocated memory pools.

Memory used for DMA operations must be *pinned*: it must have a fixed physical address and cannot be paged to the disk. Pinned memory is more expensive to maintain and has a stricter quota than normal heap memory. While a driver can allocate DMA memory at any time, that memory is only freed when the driver exits. To allow an active driver to free DMA memory, the RVM would need to ensure the device will not access the memory in the future. Freeing DMA memory also leads to fragmentation, which makes all subsequent checks of pointers to DMA memory more expensive. We chose to allow freeing DMA memory upon driver exit (after the device has been reset) for simplicity and performance. Fortunately, in practice, all the Linux drivers we ported except the USB controller driver already behave this way; we easily modified the USB driver to do the same.

Mutual exclusion. Linux drivers synchronize concurrent invocations from clients using locks, which Nexus also provides. However, Linux drivers typically synchronize with devices by disabling interrupts. While interrupts are disabled, the driver cannot be interrupted by other drivers or by the kernel. But making this same functionality available for untrusted user-space drivers allows starvation attacks.

Fortunately, typical drivers need only non-reentrant code sections, which we implement by deferring the driver's interrupts and pausing its other threads. When a driver thread enters a non-reentrant section, the Nexus scheduler marks all other threads associated with the driver as not runnable; the kernel and other processes are unaffected. Interrupts for this driver are delayed until it finishes the non-reentrant section, as they would be with interrupts disabled in hardware.[7] In this approach, the driver does not have exclusive control of the CPU, but it avoids being called in a reentrant manner by concurrent invocations or by interrupts.

Our implementation of deferred interrupts may cause problems for drivers that require precise timing. For example, the Linux i810 sound card driver calibrates playback speed by measuring playback progress over a fixed-length period during initiation. Such precise scheduling can be viewed as a privilege that drivers do not need. We rewrote the sound driver to measure the interval over which its calibration routine ran rather than using a fixed-length period; precisely measuring time in user space requires no special privileges.

Driver	Linux LoC	Lines changed	Lines added	DSS LoC
i810	5,500	26	56	149
e1000	11,849	50	3	303
USB UHCI	13,328	169	525	508
USB mouse	650	6	16	-
USB disk	19,767	29	121	-

Figure 2: Lines of code in each ported Linux driver and DSS. USB mouse and disk drivers are monitored by the UHCI DSS.

5 Results

We implemented user-space device drivers for the i810 sound card, e1000 network card, USB UHCI controllers, USB mice, and USB disks in Nexus. Here, we quantify the performance, robustness, and complexity of these drivers, their DSSes, and the Nexus RVM.

We quantify the ease of driver porting and the auditability of DSSes by counting the number of lines of code in each DSS and the number of lines changed to port each Linux driver to Nexus. These counts are given in Figure 2. We distinguish between lines we modified in the Linux driver files and lines we added in new files. The number of changed and added lines was small, and as expected, each DSS is dramatically smaller than the corresponding driver. Our DSSes are similar in size to descriptions of network devices in Devil [25] and to the safety annotations applied to drivers in Spec# [8].

We wrote each DSS by referring to the manufacturer's documentation about device behavior and to existing drivers. The DSS for USB UHCI was derived entirely from the documentation. The i810 and e1000 DSSes are based on documentation that describes features our drivers actually use; other features are disallowed by the DSS. Writing a DSS based on an existing driver is tempting, but risks disqualifying other drivers that attempt different (but safe) behavior. Writing a DSS based on all features described in published documentation is more time-consuming, but in theory, it admits any legal driver. Based on our experience, we estimate the time to develop a DSS, given a working driver, manufacturer's documentation, and familiarity with the DSS language but not with the device, as one to five days.

5.1 Driver performance

To gain insight into the performance of our user-space device drivers, we tested each at idle and under load. Our test system was a 3.0 GHz Pentium 4 system dual-booting Nexus and Linux 2.4.22. For network tests, the remote host was a 2.4 GHz Athlon 64 X2 system running

Linux 2.6.22, connected over a switched, lightly loaded 1 Gbps network.

To obtain a detailed breakdown of the sources of overhead, we instrumented several versions of the e1000 network driver and the i810 sound driver:

- **Linux**: An in-kernel Linux driver.
- **Kernel**: An in-kernel Nexus driver.
- **Unsafe**: A Nexus user-space driver, but with no reference monitor. This driver has direct access to I/O and DMA.
- **Nullspec**: A monitored Nexus user-space driver but with the trivial reference monitor, which is satisfied by any sequence of events.
- **Safe**: A driver with a full reference monitor.

These driver versions specifically quantify the costs of running under Nexus (Kernel), running in user space (Unsafe), monitoring I/O and DMA operations (Nullspec), and checking operations against a specification (Safe). Overall, these drivers permit us to apportion the costs of safe user-space drivers to the various mechanisms needed to support them.

The Unsafe, Nullspec, and Safe drivers for the e1000 include some simple optimizations:

- We changed monitored DMA memory accesses from dereferences (i.e., page faults) to explicit system calls.
- We combined sequences of unconditional reads or writes into a single system call. The driver writes between 8 and 2,048 bytes in a logical operation. Normally, these are written 4 bytes at a time; we added a system call to handle a sequence as one operation.
- We stored in the driver the result of reads from a status register. The driver reads the register repeatedly to check several bits. It does not need (and is not expecting) fresh values each time. Thus, we combined several nearby reads into a single system call.

We determined where to apply these techniques by identifying code in the driver that most often called `read` and `write` system calls and caused page faults. We changed 39 lines of driver code (in less than half a day), with dramatic results: we nearly doubled the receive bandwidth and nearly tripled the packet processing rate. Figure 3 shows the effect of the optimizations when receiving 1470-byte packets. All of the measurements below also include these optimizations.

To test bulk data throughput of the e1000 driver, we sent UDP packets at 1 Gbps to and from a Linux host running Iperf [32]. We varied the size of each packet from 100 bytes to 1470, in order to find the limits of packet-processing rate and data rate. Figures 4 and 5 show the performance, in Mbps and in thousands of packets per second, for all versions of the e1000 driver. All five versions of the e1000 driver performed identically when receiving packets. The three user-space drivers—Unsafe, Nullspec, and Safe—show somewhat degraded performance when sending packets smaller than 800 bytes. The user-space drivers take longer to handle interrupts, and sending generates more interrupts than receiving because the e1000 driver receives (but does not send) many packets per interrupt under heavy load.

Optimizations	Packets/sec	Throughput
Page faults	43,203	511.6 Mbps
Syscalls	65,074	753.5 Mbps
Syscalls+batching+caching	123,328	947.7 Mbps

Figure 3: Performance effects of replacing page faults with system calls, then batching and caching groups of operations.

To measure interrupt handling times, we instrumented the interrupt handler for the i810 driver. This test uses the CPU cycle counter for nanosecond timing, with instrumentation added to the kernel's trap function (where an interrupt is first visible to software) and to the exit point of the interrupt handler. Average interrupt processing time, over 120 samples, was $5.3 \pm 0.2\mu$s for Linux, $8.5 \pm 0.2\mu$s for Kernel, $22.1 \pm 1.5\mu$s for Unsafe, $37.9 \pm 2.4\mu$s for Nullspec, and $46.9 \pm 3.8\mu$s for Safe. So, the user-space interrupt handlers took three to five times as long as the in-kernel Nexus drivers. This slowdown is not unexpected, because user-space handlers require a scheduler invocation and two or more context switches.

A macrobenchmark for network round-trip time, which includes driver response time, is the `ping` command, which sends an ICMP echo request packet and receives an ICMP echo reply packet in return. The replies are normally generated by the remote kernel, resulting in low latencies. The elapsed time between sending the request and receiving the reply is the network round-trip time plus the time required for the remote host to process the request. We measured ping times from a Linux box to a Nexus box running each of the four test e1000 drivers. The average round-trip time, over 100 packets, was $103 \pm 35\mu$s for Kernel, $139 \pm 41\mu$s for Unsafe, $158 \pm 55\mu$s for Nullspec, and $156 \pm 54\mu$s for Safe.

Another important driver performance metric is the CPU time spent in drivers while performing a high-level task. To quantify this, we streamed video (with audio) over HTTP and played it using `mplayer`. The video averaged 1071 Kbps and lasted for 30 seconds. The re-

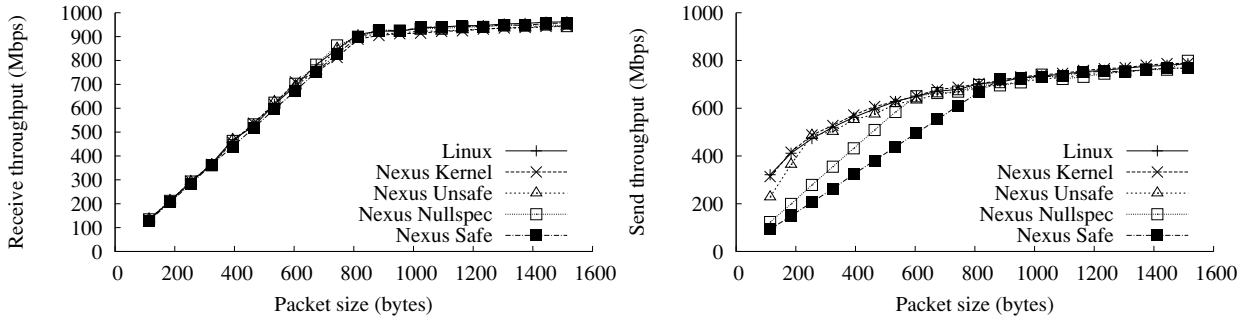


Figure 4: Throughput (Mbps) sent and received by all versions of the e1000 driver using Iperf.

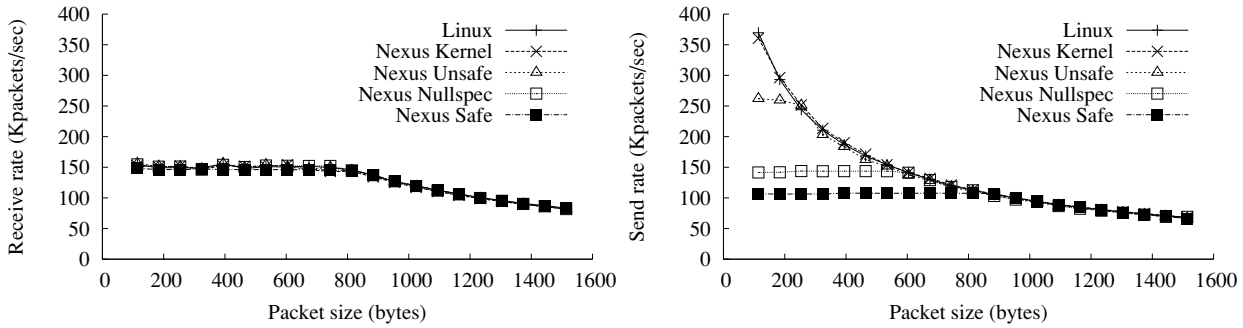


Figure 5: Throughput (thousands of packets/second) sent and received by all versions of the e1000 driver using Iperf.

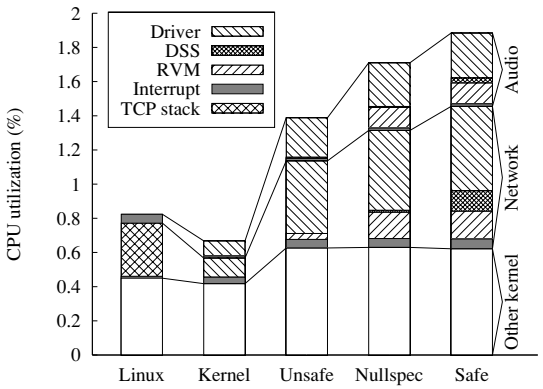


Figure 6: CPU time apportionment when streaming video over the network.

sulting CPU time spent in the network driver, the audio driver, and the kernel is shown in Figure 6. The CPU time spent in the Safe driver was about 2.5x the CPU time spent in the Kernel driver, which could be a limiting factor on more heavily loaded systems. A fair comparison of the CPU time of the Linux and Nexus kernel drivers was not possible, because TCP/IP time is included in the kernel in Linux and in a library in Nexus and cannot be factored out in either case. We believe that the CPU cost in Linux, ignoring the cost of TCP/IP, would be comparable to the CPU cost in the Nexus kernel.

We measured how often each driver executes basic operations and what each basic operation costs. The frequencies of memory, port I/O, MMIO, and interrupts are shown for each driver in Figure 7. All figures are the average rate per second when the driver is idle or under load, as indicated. For this test, the network load was a flood `ping`. Counting unmonitored memory operations (by making them monitored) makes the e1000 too slow for our tests. Hence, we estimated the rate of unmonitored memory operations for the e1000 by measuring a heavily instrumented driver under partial load, scaling its results up to what they would have been given full load.

Unmonitored memory operations are anywhere from two to 100 times more frequent than monitored memory operations, depending on the driver. We measured the average cost, over 100,000 tests, of an unmonitored memory operation as 0.59ns, a monitored memory operation executed as a system call as 0.84μs, and a monitored memory operation that causes a page fault as 1.53μs. Page faults are more expensive because they must save more state and because the page fault handler must disassemble and interpret the faulting instruction.

The cost of each basic I/O operation varies relatively little. However, the cost of checking operations against

	Audio (playback)	Network (idle)	Network (load)	USB (idle)	USB (mouse)	USB (disk)
Unmonitored mem	8018	0	4578113*	8535	19159	223346
Monitored mem	78.3	5.6	42459	0	1930	103374
Port I/O	279	0	0	267	764	956
Interrupts	15.7	1.1	2079	0	124	138
MMIO	0	139	10586	0	0	0

Figure 7: Average rate (per second) of read and write operations during steady-state operation. (* estimated result)

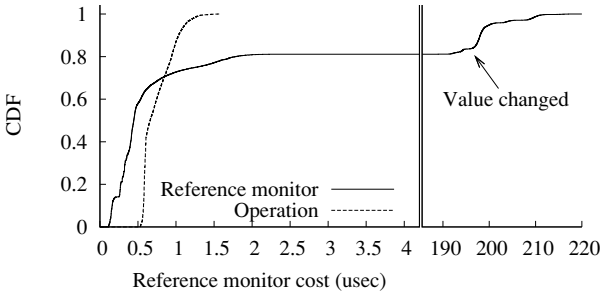


Figure 8: Cost of executing and checking USB disk port I/O operations.

	Driver			
Failure type	Nullspec		Safe	
No failure	7	(23%)	7	(1%)
Driver exits	7	(23%)	16	(1%)
RVM terminates driver		—	1132	(94%)
Driver out of sync	16	(52%)	45	(4%)
Hardware damaged	1	(3%)	0	(0%)
Total perturbation tests	31	(100%)	1200	(100%)

Figure 9: Perturbation testing results: how the Nullspec and Safe drivers failed, if at all, in repeated tests. Nullspec testing was aborted when it damaged the device.

the reference monitor can vary dramatically. Figure 8 shows the cost of checking USB port I/O operations (for disk I/O) against the reference monitor. We found that 80% of the time, the cost is under 2μs. The other 20% of the time, the cost is 190μs or more. The expensive operation is a safety check, required when the value read from a certain register changes ("value changed" in Figure 8), which happens once per millisecond. Without significant optimization, this level of overhead is likely to be too high for EHCI (high-speed USB 2.0) devices, which support nominal data rates 40 times higher than UHCI.

5.2 Driver robustness

Accepted quantitative metrics for the security of a system do not exist. Nevertheless, to establish the security of our RVM and reference monitors, we used two approaches others have used. First, we simulated unanticipated malicious drivers by randomly perturbing the interactions between drivers and the RVM, resulting in potentially invalid operations being submitted to the reference monitor and possibly to the device. Second, we built specific drivers that perpetrate known attacks on the kernel using interrupt and DMA capabilities.

We simulated unanticipated malicious drivers by changing operations and operands in a layer interposed between a legal driver and the RVM. This layer modified each operation according to an independent probability of 1 in 16,384.[8] Each operation was a read or a write; our modifications involved replacing either the address, the length, or the value (at random) with another value in the appropriate range. So, a write to an I/O port was replaced with a write to a port in the same range, a write of a different length, or a write of another value. Reads were perturbed similarly. Note, this approach does not produce repeatable experiments, because driver behavior depends on external factors like the OS scheduler and the arrival times of packets, which are not under our control.

This *perturbation testing* is similar to fuzz testing [26, 31], except that our code perturbed only I/O operations—not source or machine code. Fuzz testing emphasizes isolation properties, whereas we tested only properties enforced by the RVM and the reference monitor.

We applied perturbation testing to the e1000 driver. When the modifications were benign, the driver showed no apparent failures. Sometimes, the driver itself detected an error (e.g., a status register read failed a sanity check) and exited cleanly. Often, the reference monitor detected an illegal operation, and the RVM terminated the driver. Finally, our perturbations sometimes caused the driver to get out of sync with the device, after which no further packets were sent or received. This does not compromise the integrity or availability of the kernel or the device, so the RVM has no obligation here.[9] Figure 9 summarizes the different cases encountered in our experiments. The Nullspec driver completed more tests with no apparent failure than the Safe driver did, because the reference monitor used for the Safe driver blocks all unknown behavior—even if it might be benign.

We hoped the perturbed Nullspec driver would cause kernel livelock, starvation, or a crash. In practice, how-

ever, the likelihood of causing driver crashes and stalls is much higher. The 31st run of the Nullspec test rendered the device unusable: neither the Linux nor the Nexus driver could thereafter initialize the card.[10] We replaced the card, but we do not plan further perturbation testing.

In addition to perturbation testing, we wrote several malicious drivers to execute specific attacks on the kernel using the e1000's interrupt and DMA capabilities:

- **Livelock:** The driver never acknowledges interrupts, resulting in a flood of interrupt activity and starvation for all other processes.
- **DMA kernel crash:** The driver uses the device to write to kernel memory, resulting in a system crash.
- **DMA kernel read:** The driver sends a sensitive page (e.g., containing a secret key) to a remote host.
- **Direct kernel read/write:** The driver constructs a pointer and reads or writes sensitive data directly.
- **DMA kernel code injection:** The driver points a DMA buffer pointer at system call code, then pings a remote machine with attack code.[11] The response is written over the target system call implementation. The attacking driver then invokes the system call to gain control of the kernel.
- **DMA read/write to other device:** The driver uses a ping to overwrite video memory, resulting in an image appearing on the screen.

Not surprisingly, the livelock and DMA attacks succeed when run as Unsafe or Nullspec drivers, all the attacks succeed as Kernel drivers, and they are all are caught by the RVM when run in Safe mode. The livelock attack is prevented by the RVM terminating any driver that does not acknowledge the interrupt by reading the interrupt control register. The DMA attacks are prevented by the RVM terminating any driver that attempts to transmit or receive packets with any invalid addresses in the transmit or receive buffer lists. Finally, any direct attempt to read or write the memory of other drivers is blocked by hardware isolation in all modes except Kernel.

6 Related Work

Several existing operating systems implement device drivers in user space for isolation or modularity, but without monitoring I/O and DMA operations. Hence, these systems do not defend against malicious operations by drivers. The Michigan Terminal System [9] on the IBM 360 architecture seems to be the earliest operating system to implement device drivers as user programs. Dijkstra's THE multiprogramming system [11] is organized into *levels*. Level 3 contains device drivers; level 0 implements a scheduler and the interrupt dispatch routine; level 2 implements semaphores, which are used to convey interrupts to device drivers. THE ran on hardware without memory protection, achieving modularity but not isolation. The SUE separation kernel [27] organizes components, including device drivers, into isolated domains akin to hosts in a distributed system. SUE uses memory protection to restrict each driver's access to I/O ports, but it provides no DMA or interrupt protection: DMA is excluded completely, and components are trusted to yield control after each interrupt or task switch.

L3 [24], MINIX 3 [19], and a modified Linux by Leslie et al. [22] all implement at least some drivers in user space, allowing each driver access to a limited set of I/O ports. This approach protects against naive attacks and at least some bugs. However, all three systems allow DMA, meaning that drivers remain trusted. Leslie includes performance results, which are comparable to the throughput and CPU overhead of our Unsafe (unmonitored) drivers.

Nooks [31] and Shadow Drivers [30] provide hardware-based isolation and fail-over operation for drivers within the Linux kernel, to prevent accidental overwriting of kernel structures. Nooks protects against common bugs, like accidental writes to memory structures belonging to another kernel component. Program rewriting techniques, such as Software-based Fault Isolation (SFI) and its successors [12, 13], implement similar isolation properties in software. SafeDrive [33] uses program annotations and lightweight run-time checks to enforce type safety and bounds checking, but is explicitly not designed to handle malicious drivers. None of these techniques restricts what I/O operations are sent to devices, though SFI could; we are pursuing this approach as future work.

Microdrivers [16, 17] are a hybrid implementation of Linux device drivers, with up to 65% of the driver running in user space and only the most performance-sensitive code remaining in the kernel. Microdrivers handle network interrupts in the kernel, so they are not secure. Their performance is comparable to the performance of Nexus Unsafe drivers.

Some operating systems take steps to prevent malicious drivers from misusing I/O ports or DMA transfers. Mungi [23] (on Alpha and Itanium platforms) and Scomp [14] (on custom hardware) use an IOMMU for DMA protection. Singularity [21, 29] enforces type-safe interactions between drivers and devices. Originally, this type safety meant unmediated access to a restricted set of ports and memory. Singularity now relies on IOMMUs to validate DMA operations, and it does not limit

interrupt rates. DROPS [18] anticipated the need for device-specific DMA monitoring prior to IOMMUs on commodity hardware.

Some safety properties can be checked statically, given rich enough rule sets or program annotations. SLAM [5] uses static rules to detect incorrect calls to the Windows driver API, but it does not enforce properties specific to any particular device. In contrast, Bierhoff and Hawblitzel extend Singularity to enforce stateful properties in SPEC# [8] much like the DMA checks in our DSSes. Static verification cannot enforce some properties a reference monitor can, especially timing properties.

Virtual machine monitors (VMMs) sometimes use drivers running in a guest operating system to control devices, instead of virtualizing all devices with drivers in the VMM. These *pass-through drivers* are inherently safe for some devices, such as USB peripherals, but not for other devices, such as disks or network cards. Xen [6, 15] puts some device drivers in *driver domains*, which are protected against most crashes but not against malicious behavior; hence, driver domains are trusted.

7 Conclusion

In traditional monolithic and microkernel operating systems, every flaw in a device driver is a potential security hole given the absence of mechanisms to contain the (mis)behavior of device drivers. We have applied the principle of least privilege to Nexus device drivers by creating an infrastructure to run these drivers in user space and by filtering their I/O operations through a reference validation mechanism (RVM). The RVM is independent of drivers and devices; device-specific information is gathered into a device safety specification (DSS) that we compile into a reference monitor. The RVM consults the reference monitor before allowing each I/O operation; any disallowed operation results in the offending driver being terminated.

An obvious question is whether or not the attacks our RVM prevents are realistic. We do not know of malicious drivers "in the wild" that use a device to escalate their privileges, although we have built several of them. The reason such drivers are not yet a real threat is probably that production systems run most drivers in the kernel and in the TCB, where violating security properties can be done directly. Systems with drivers in user space are increasingly common and will inspire attacks through devices. Our RVM and DSS can prevent these attacks.

Acknowledgments. We are grateful to Mike Swift for feedback on a draft of this work.

References

[1] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel virtualization technology for directed I/O. *Intel Technology Journal*, 10(3), Aug. 2006.

[2] B. B. Alessandro Forin, David Golub. An I/O system for Mach. In *Proceedings of the USENIX Mach Symposium*, Monterey, CA, Nov. 1991.

[3] J. P. Anderson. Computer security technology planning study—Vol. II. Technical Report ESD-TR-73-51, Electronic Systems Division, AFSC, L. G. Hanscom Field, Bedford, MA, Sept. 1972.

[4] S. Apiki. I/O virtualization and AMD's IOMMU. `http://developer.amd.com/documentation/articles/pages/892006101.aspx`, Aug. 2006.

[5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proceedings of EuroSys*, Leuven, Belgium, Apr. 2006.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of SOSP*, Bolton Landing, NY, Oct. 2003.

[7] B. N. Bershad, S. Savage, D. Becker, M. Fiuczynski, and E. G. Sirer. Protection is a software issue. In *Proceedings of HotOS*, 1995.

[8] K. Bierhoff and C. Hawblitzel. Checking the hardware-software interface in Spec#. In *Proceedings of PLOS*, Stevenson, WA, Oct. 2007.

[9] D. W. Boettner and M. T. Alexander. The Michigan Terminal System. *Proceedings of the IEEE*, 63(6):912–918, June 1975.

[10] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *Proceedings of SOSP*, Banff, Canada, Oct. 2001.

[11] E. W. Dijkstra. The structure of the 'THE'-multiprogramming system. *Communications of the ACM*, 11(5):341–346, 1968.

[12] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proceedings of OSDI*, Seattle, WA, Nov. 2006.

[13] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*, Caledon Hills, Ontario, Canada, Sept. 1999.

[14] L. J. Fraim. Scomp: A solution to the multilevel security problem. *Computer*, 16(7):26–34, 1983.

[15] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of The 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, Boston, MA, Oct. 2004.

[16] V. Ganapathy, A. Balakrishnan, M. M. Swift, and S. Jha. Microdrivers: A new architecture for device drivers. In *Proceedings of HotOS*, San Diego, CA, May 2007.

[17] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *Proceedings of ASPLOS*, Seattle, WA, Mar. 2008.

[18] H. Härtig, J. Löser, F. Mehnert, L. Reuther, M. Pohlack, and A. Warg. An I/O architecture for microkernel-based operating systems. Technical Report TUD-FI03-08, Dresden University of Technology Department of Computer Science, D-01062 Dresden, Germany, July 2003.

[19] J. N. Herder, H. Bos, and A. S. Tanenbaum. A lightweight method for building reliable operating systems despite unreliable device drivers. Technical Report IR-CS-018, Vrije Universiteit, Amsterdam, The Netherlands, Jan. 2006.

[20] M. Hirzel and R. Grimm. Jeannie: Granting Java Native Interface developers their wishes. In *Proceedings of OOPSLA*, Montréal, Canada, Oct. 2007.

[21] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, Redmond, WA, Oct. 2005.

[22] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science & Technology*, 20(5):654–664, Sept. 2005.

[23] B. Leslie and G. Heise. Towards untrusted device drivers. Technical Report UNSW-CSE-TR-0303, University of New South Wales, Sydney, Australia, Mar. 2003.

[24] J. Liedtke, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, and G. Szalay. Two years of experience with a μ-kernel based OS. *SIGOPS Oper. Syst. Rev.*, 25(2):51–62, 1991.

[25] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Proceedings of OSDI*, San Diego, CA, 2000.

[26] W. T. Ng and P. M. Chen. The systematic improvement of fault tolerance in the Rio file cache. In *Proceedings of the IEEE Symposium on Fault-Tolerant Computing (FTCS)*, Madison, WI, June 1999.

[27] J. Rushby. The design and verification of secure systems. In *Proceedings of SOSP*, Asilomar, CA, Dec. 1981.

[28] A. Shieh, D. Williams, E. G. Sirer, and F. B. Schneider. Nexus: A new operating system for trustworthy computing (extended abstract). In *Proceedings of SOSP*, Brighton, UK, Oct. 2005.

[29] M. Spear, T. Roeder, O. Hodson, G. C. Hunt, and S. Levi. Solving the starting problem: Device drivers as self-describing artifacts. In *Proceedings of EuroSys*, Leuven, Belgium, Apr. 2006.

[30] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems*, 24(4):333–360, Nov. 2006.

[31] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1):77–110, Feb. 2005.

[32] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf: The TCP/UDP bandwidth measurement tool. `http://dast.nlanr.net/Projects/Iperf`, May 2005.

[33] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: safe and recoverable extensions using language-based techniques. In *Proceedings of OSDI*, Seattle, WA, Nov. 2006.

Notes

[1] Some drivers, such as the clock, provide functionality needed for defining or enforcing security policies. These device drivers remain part of the TCB no matter where they execute.

[2] In our experience, these identifiers are sufficient. Three additional PCI ID fields are available, but our DSS selection code does not depend on them.

[3] As an extension to our work, we have considered a composite approach to writing DSSes: the composite DSS is derived from the controller DSS and an auxiliary DSS for each attached device.

[4] Some predicates and actions are too complex to write in terms of the simple syntax currently supported by our DSS language, where user-defined state variables must be scalars, and predicates cannot be recursive. The DSS compiler therefore supports embedded blocks of C, coded as `C:{...}`, appearing in predicates and in actions. Within an embedded C block, it is possible to nest an embedded block of DSS code, e.g., to use an identifier or an operator not available in C. Our syntax was inspired by Java and C nesting in Jeannie [20].

[5] This timeout is the only input to the reference monitor that does not come from either the driver or the device. It comes from the kernel.

[6] Linux 2.4.22, though not current, is the version on which parts of Nexus are based. We used drivers from this version of Linux to simplify implementation.

[7] This technique would be both correct and efficient on multiprocessor systems, although Nexus does not yet run on multiprocessors.

[8] We also tried higher and lower probabilities, resulting in more and fewer errors than reported here.

[9] The RVM does not attempt to prevent incorrect or incomplete service (see Section 3.1).

[10] Would the reference monitor have prevented the damage if it had been enabled for that test? We cannot be sure due to the inherent nondeterminism of peripheral devices, but we believe it would have. We ran 1200 reference-monitored tests with no damage to the device.

[11] The e1000 can retrieve any physical memory location by DMA and send it as a network packet, or it can overwrite any physical memory location with the contents of incoming packets. It cannot directly transfer one memory page to another. To get around this, we use `ping` packets; most other hosts will echo a packet with arbitrary contents, which enables us to copy from one local memory location to another by way of a remote host.

Appendix: DSS Example

The following is an abridged version of our DSS for the Intel i810 audio device. It defines the device ID, followed by the state variables and a reset routine. A *NAMES* section then introduces labels for the various events associated with I/O register operations and interrupts. Finally, a *TRANSITIONS* section defines the allowed transitions for the state machine. By default, upon receipt of an input, all transitions are checked, and actions are applied (in unspecified order) for each satisfied predicate. Inside an **ordered** block, transitions are checked sequentially only until a predicate is matched; at most one action is applied inside the block. Several transitions in this DSS have empty actions—they accept an input without changing the state of the state machine.

```
hardware: "PCI:8086:24d5";
monitored region $RING_DMA;        // Define a monitored region to contain DMA descriptors.
const $RING_LEN = 8 * 32;
var $DMA_ENABLED = 0;              // Define a state variable: true when device DMA is active.
reset: C:{                         // Restore device to state with no DMA or interrupts.
   outb(0, $PORTIO[1].base + $CONTROL_OFFSET);          // Turn off playback DMA.
   while(inb($PORTIO[1].base + $CONTROL_OFFSET) != 0) ; // Wait for acknowledgment.
   $DMA_ENABLED = 0;
}

//*************** NAMES ******************
// Each line maps write, read, and read_response operations on a register (address, size) to a logical name.
// Syntax: <offset, length> --> write_name, read_name, read_response_name;
names for $PORTIO[1], $MMIO[1]:
// Writes to base+0x10 with size=4 are known as write_playback_dma_base.
<0x10, 4> --> write_playback_dma_base($VAL), safe, safe;
<0x16, 1> --> write_status($VAL), safe, read_response_status($VAL);
<0x1b, 1> --> write_control($VAL), safe, safe;                  // Reading the control register is always allowed.
names for $RING_DMA mod 8:                                     // Define names for writes to DMA descriptors.
<0x00, 4> --> write_descriptor_base($ADDR, $VAL), safe, safe;  // offsets 0, 8, 16, ...
<0x04, 4> --> write_descriptor_len($ADDR, $VAL), safe, safe;   // offsets 4, 12, 20, ...
names for $INTR[0]:
 * --> i810_intr;                                              // The only interrupt is named i810_intr.

//*************** TRANSITIONS **************
// Syntax: P_i { A_i }
// Modifying the DMA base register is only allowed if DMA is not running and the address points to monitored memory.
write_playback_dma_base(val) && $DMA_ENABLED == 0 && exists($MONITORED[i]) suchthat
   range(val, $RING_LEN) in $MONITORED[i] { $RING_DMA = range(val, $RING_LEN); }

// Starting DMA is allowed only when the DMA base register points to 32 pointers to pinned, unmonitored memory.
write_control(val) && (val & 0x01) == 1 && $RING_DMA != null && (forall(k) = 0..31 (exists($UNMONITORED[j])
   suchthat range(fetch($RING_DMA.base + 8*k, 4), fetch($RING_DMA.base + 8*k+4, 2)) in $UNMONITORED[j]))
   { $DMA_ENABLED = 1; }
write_control(val) && (val & 0x01) == 0 { $DMA_ENABLED = 0; }

// Changing DMA descriptors is legal if DMA is inactive, or if the modified entry points to pinned, unmonitored memory.
write_descriptor_base(addr, val) && ($DMA_ENABLED == 0) {}
write_descriptor_base(addr, val) && ($DMA_ENABLED != 0) &&
   (exists($UNMONITORED[j]) suchthat range(val, fetch(addr + 4, 2)) in $UNMONITORED[j]);
write_descriptor_len(addr, val) && ($DMA_ENABLED == 0) {}
write_descriptor_len(addr, val) && ($DMA_ENABLED != 0) &&
   (exists($UNMONITORED[k]) suchthat range(fetch(addr - 4, 4), bits(val, 0..15)) in $UNMONITORED[k]);

// The i810 interrupt acknowledgment protocol: first, the driver checks if the interrupt came from i810 by reading status bits 2..4;
// then, if so, acknowledges it by writing status bits 2..4.
ordered {                          // In an "ordered" block, transitions are checked only until the first match.
   read_response_status(val) && bits(val, 2..4) == 0 { $INTR[0].status = idle; }   // i810 is not asserting an interrupt.
   read_response_status(val) {}                                                    // Otherwise interrupt is still pending.
}
write_status(val) && bits(val, 2..4) != 0 { $INTR[0].status = idle; }              // Acknowledging interrupts is legal.

i810_intr <16, 1, 1> {}            // Interrupt is rate-limited to 16 per second, no bursts.
```

Digging For Data Structures

Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King
University of Illinois at Urbana-Champaign

Abstract

Because writing computer programs is hard, computer programmers are taught to use encapsulation and modularity to hide complexity and reduce the potential for errors. Their programs will have a high-level, hierarchical structure that reflects their choice of internal abstractions. We designed and forged a system, Laika, that detects this structure in memory using Bayesian unsupervised learning. Because almost all programs use data structures, their memory images consist of many copies of a relatively small number of templates. Given a memory image, Laika can find both the data structures and their instantiations.

We then used Laika to detect three common polymorphic botnets by comparing their data structures. Because it avoids their code polymorphism entirely, Laika is extremely accurate. Finally, we argue that writing a data structure polymorphic virus is likely to be considerably harder than writing a code polymorphic virus.

1 Introduction

System designers use abstractions to make building complex systems easier. Fixed interfaces between components allow their designers to innovate separately, reduce errors, and construct the complex computer systems we use today. The best interfaces provide exactly the right amount of detail, while hiding most of the implementation complexity.

However, no interface is perfect. When system designers need additional information they are forced to bridge the gap between levels of abstraction. The easiest, but most brittle, method is to simply hard-code the mapping between the interface and the structure built on top of it. Hard-coded mappings enable virtual machine monitor based intrusion detection [13, 22] and discovery of kernel-based rootkits using a snapshot of the system memory image [29]. More complicated but potentially more robust techniques infer details by combining general knowledge of common implementations and runtime probes. These techniques allow detection of OS-level processes from a VMM using CPU-level events [18, 20], file-system-aware storage systems [19, 31], and storage-aware file systems [28]. Most of these techniques work because the interfaces they exploit can only be used in a very limited number of ways. For example, only an extremely creative engineer would use the CR3 register in an x86 processor for anything other than process page tables.

The key contribution of this paper is the observation that even more general interfaces are used often by programmers in standard ways. Because writing computer programs correctly is so difficult, there is a large assortment of software engineering techniques devoted to making this process easier and more efficient. Ultimately most of these techniques revolve around the same ideas of abstraction and divide-and-conquer as the original interfaces. Whether this is the only way to create complex systems remains to be seen, but in practice these ideas are pounded into prospective programmers by almost every text on computer science, from *The Art of Computer Programming* to the more bourgeoisie *Visual Basic for Dummies*.

We chose to exploit a small piece of this software engineering panoply, the compound data structure. Organizing data into objects is so critical for encapsulation and abstraction that even programmers who do not worship at the altar of object-oriented programming usually use a significant number of data structures, if only to implement abstract data types like trees and linked lists. Therefore we can expect the memory image of a process to consist of a large number of instantiations of a relatively small number of templates.

This paper describes the design and implementation of a system – which we named Laika in honor of the Russian space dog – for detecting those data structures given a memory image of the program. The two key

challenges are identifying the positions and sizes of objects, and determining which objects are similar based on their byte values. We identify object positions and sizes by using potential pointers in the image to estimate object positions and sizes. We determine object similarity by converting objects from sequences of raw bytes into sequences of semantically valued blocks: "probable pointer blocks" for values that point into the heap or the stack, "probable string blocks" for blocks that contain null-terminated ASCII strings, and so on. Then, we cluster objects with similar sequences of blocks together using Bayesian unsupervised learning.

Although conceptually simple, detecting data structures in practice is a difficult machine learning problem. Because we are attempting to detect data structures without prior knowledge, we must use unsupervised learning algorithms. These are much more computationally complex and less accurate than supervised learning algorithms that can rely on training data. Worse, the memory image of a process is fairly chaotic. Many malloc implementations store chunk information inside the chunks, blending it with the data of the program. The heap is also fairly noisy: a large fraction consists of effectively random bytes, either freed blocks, uninitialized structures, or malloc padding. Even the byte/block transformation is error-prone, since integers may have values that "point" into the heap. Despite these difficulties, Laika manages reasonable results in practice.

To demonstrate the utility of Laika, we built a virus checker on top of it. Current virus checkers are basically sophisticated versions of grep [2]. Each virus is identified with a fingerprint, usually a small sequence of instructions. When the virus checker finds that fingerprint in a program, it classifies it as a version of the corresponding virus. Because it is easy to modify the instruction stream of a program in provably correct ways, virus writers have created polymorphic engines that replace one set of instructions with another computationally equivalent one, obfuscating the fingerprints [25]. Most proposals to combat polymorphic viruses have focused on transforming candidate programs into various canonical formats in order to run fingerprint scanners [5, 8, 23].

Instead, our algorithm classifies programs based on their data structures: if an unknown program uses the same data structures as Agobot, it is likely to in fact be a copy of Agobot. Not only does this bypass all of the code polymorphism in current worms, but the data structures of a program are likely to be considerably more difficult to obfuscate than the executable code – roughly compiler-level transformations, rather than assembler-level ones. Our polymorphic virus detector based on Laika is over 99% accurate, while ClamAV, a leading open source virus detector, manages only 85%. Finally, by detecting programs based on completely different features our detector has a strong synergy with traditional code-based virus detectors.

Memory-based virus detection is especially effective now that malware writers are turning from pure mayhem to a greedier strategy of exploitation [1, 12]. A worm that merely replicates itself can be made very simple, to the point that it probably does not use a heap, but a botnet that runs on an infected computer and provides useful (at least to the botnet author) services like DoS or spam forwarding is more complex, and more complexity means more data structures.

2 Data Structure Detection

A classifier is an algorithm that groups unknown objects, represented by vectors of features, into semantic classes. Ideally, a classification algorithm is given both a set of correctly classified training data and a list of features. For example, to classify fruit the algorithm might be given the color, weight, and shape of a group of oranges, apples, watermelons, and bananas, and then asked whether a 0.1 kg red round fruit is an apple or a banana. This is called supervised learning; a simple example is the naive Bayes classifier, which learns a probability distribution for each feature for each class from the training data. It then computes the class membership probability for unknown objects as the product of the class feature probabilities and the prior probabilities, and places the object in the most likely class. When we do not have training data, we must fall back to unsupervised learning. In unsupervised learning, the algorithm is given a list of objects and features and directed to create its own classes. Given a basket of fruit, it might sort them into round orange things, round red things, big green things, and long yellow things. Designing a classifier involves selecting features that expose the differences between items and algorithms that mirror the structure of the problem.

2.1 Atoms and Block Types

The most important part of designing any classifier is usually selecting the features. Color, shape, and weight will work well for fruit regardless of what algorithm is used, but country of origin will not. This problem is particularly acute for data structure detection, because two objects from the same class may have completely different byte values. Our algorithm converts each machine word (4 on 32-bit machines, 8 bytes on 64-bit machines) into a *block type*. The basic block types are address (points into heap/stack), zero, string, and data (everything else). This converts objects from vectors of bytes

Address	Value	Char Value	Block
0x650000	0x20	“!”	D
0x650008	0x0	“\0”	0
0x650010	0x650028	“\FS\0e”	A
0x650018	0x650088	“\^\0e”	A
0x650020	0x20	“!”	D
0x650028	0x650008	“\BS\0e”	A
0x650030	0x650048	“0\0e”	A
0x650038	0x650068	“h\0e”	A
0x650040	0x20	“!”	D
0x650048	0x650028	“\FS\0\e”	A
0x650050	0x0	“\0”	0
0x650058	0x650068	“h\0e”	A
0x650060	0x20	“!”	D
0x650068	0x6873696620656E6F	“one fish”	S
0x650070	0x6966206F7774202C	“, two fi”	S
0x650078	0x20646572202C6873	“sh, red ”	S
0x650080	0x20	“!”	D
0x650088	0x6C62202C68736966	“fish, bl”	S
0x650090	0x2E68736966206575	“ue fish.”	S
0x650098	0x56700	“\0g\ENQ”	D
0x6500A0	0x40	“A”	D

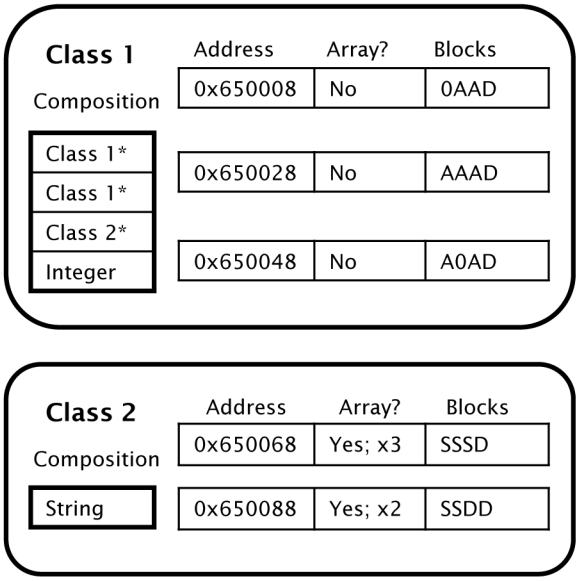


Figure 1: An example of data structure detection. On the left is a small segment of the heap, and on the right is Laika's output. Class 1, in rows 2-13, is a doubly linked list of C strings; the first two elements are pointers to other elements of Class 1. The fourth element is actually internal malloc data on the size of the next chunk. Laika estimates the start of the next object as the end of the first, which is 8 bytes too long. Class 2 contains two C null-terminated character arrays. A real heap sample is much noisier; in the programs we looked at, less than 50% of the heap was occupied by active objects; the rest was a mix of freed objects, malloc padding, and unallocated chunks.

into vectors of block types, and we can expect the block type vectors to be similar for objects from the same class.

Classes are represented as vectors of *atomic types*. Each atomic type roughly corresponds to one block type (e.g., pointer → address and integer → data), but there is some margin for error since the block type classification is not always accurate; some integer or string values may point into the heap, some pointers may be uninitialized, a programmer may have used a union, and so on. This leads to a probability array $p(blocktype|atomictype)$ where the largest terms are on the diagonal, but all elements are nonzero. While these probabilities will not be exactly identical for individual applications, in our experience they are similar enough that a single probability matrix suffices for most programs. In the evaluation we measured $p(blocktype|atomictype)$ for several programs; all of them had strongly diagonal matrices. There is also a random atomic type for blocks that lie between objects. Figure 1 shows an example of what memory looks like when mapped to block and atomic types.

Although the block type/atomic type system allows us to make sense of the otherwise mystifying bytes of a memory image, it does have some problems. It will miss unaligned pointers completely, since the two halves of the pointer are unlikely to point at anything useful, and it will also miss the difference between groups of integers, for example four 2-byte integers vs. one 8-byte integer. In our opinion these problems are relatively minor, since unaligned pointers are quite rare, and it would be extremely difficult to distinguish between four short ints and one long int anyway. Lastly, if the program occupies a significant fraction of its address space there will be many integer/pointer mismatches, but as almost all new CPUs are 64-bit, the increased size of a 64-bit address space will reduce the probability of false pointers.

2.2 Finding Data Structures

Unlike the idyllic basket of fruit, it is not immediately obvious where the objects lurk in an image, but we can estimate their positions using the targets of the pointers. Our algorithm scans through a memory image looking for all pointers, and then tentatively estimates the positions of objects as those addresses referenced in other places of the image.

Although pointers can mark the start of an object, they

seldom mark the end. Laika estimates the sizes of objects during clustering. Each object's size is bounded by the distance to the next object in the address space, and each class has a fixed size, which is no larger than its smallest object. If an object is larger than the size of its class, its remaining blocks are classified as random noise. For this purpose we introduce the random atomic type, which generates all block types more or less equally. In practice some objects might be split in two by internal pointers - pointers to the interior of malloc regions. At the moment we do not merge these objects.

2.3 Exploiting Malloc

It should be possible for Laika to find objects more accurately when armed with prior information about the details of the malloc library. For example, the Lea allocator, used in GNU libc, keeps the chunk size field inlined at the top of the chunk as shown in figure 1. Unfortunately there is sufficient variety in malloc implementations to make this approach tedious. Aside from separate malloc implementations for Windows and Linux, there are many custom allocators, both for performance and other motivations like debugging.

Early versions of Laika attempted to exploit something that most malloc implementations should share: address space locality. Most malloc implementations divide memory into chunks, and chunks of the same size often lie in contiguous regions. Since programs exhibit spatial locality as well, this means that an object will often be close to one or more objects of the same type in memory; in the applications we measured over 95% of objects had at least one object of the same class within their 10 closest neighbors. Despite this encouraging statistic, the malloc information did not significantly change Laika's classification accuracy. We believe this occurs because Laika already knows that nearby objects are similar due to similar size estimations.

2.4 Bayesian Model

Bayesian unsupervised learning algorithms compute a joint probability over the classes and the classification, and then select the most likely solution. We represent the memory image by $\boldsymbol{M}$, where $\boldsymbol{M}_l$ is the lth machine word of the memory image, $\boldsymbol{\omega}$ for the list of classes, where $\boldsymbol{\omega}_{jk}$ is the kth atomic type of class j, and $\boldsymbol{X}$ for the list of objects, where $\boldsymbol{X}_i$ is the position of the ith object. We do not store the lengths of objects, since they are implied by the classification. Our notation is summarized in Table 1.

We wish to estimate the most likely set of objects and classes given a particular memory image. With Bayes's rule, this gives us:

$$p(\boldsymbol{\omega}, \boldsymbol{X}|\boldsymbol{M}) = \frac{p(\boldsymbol{M}|\boldsymbol{\omega}, \boldsymbol{X})p(\boldsymbol{X}|\boldsymbol{\omega})p(\boldsymbol{\omega})}{p(\boldsymbol{M})} \quad (1)$$

Since Laika is attempting to maximize $p(\boldsymbol{\omega}, \boldsymbol{X}|\boldsymbol{M})$, we can drop the normalizing term $p(\boldsymbol{M})$, which does not affect the optimum solution, and focus on the numerator. $p(\boldsymbol{\omega})$ is the prior probability of the class structure. To simplify the mathematics, we assume independence both between and within classes, even though this assumption is not really accurate (classes can often contain arrays of basic types or even other objects). We let $\boldsymbol{\omega}_{jk}$ be the kth element of class j, which lets us simply multiply out over all such elements:

$$p(\boldsymbol{\omega}) = \prod_j \prod_k p(\boldsymbol{\omega}_{jk}) \quad (2)$$

$p(\boldsymbol{X}|\boldsymbol{\omega})$ is the probability of the locations and sizes of the list of objects, based on our class model and what we know about data structures. This term is 0 for illegal solutions where two objects overlap, and 1 otherwise. $p(\boldsymbol{M}|\boldsymbol{\omega}, \boldsymbol{X})$ represents how well the model fits the data. When the class model and the instantiation list are merged, they predict a set of atomic data types (including the random "type") that cover the entire image. Since we know the real block type $\boldsymbol{M}_l$, we can compute the probability of each block given classified atomic type:

$$p(\boldsymbol{M}|\boldsymbol{\omega}, \boldsymbol{X}) = \prod_l p(\boldsymbol{M}_l|\boldsymbol{\omega}, \boldsymbol{X}) \quad (3)$$

When $p(\boldsymbol{\omega})$, $p(\boldsymbol{X}|\boldsymbol{\omega})$, and $p(\boldsymbol{M}|\boldsymbol{\omega}, \boldsymbol{X})$ are multiplied together, we finally get a master equation which we can use to evaluate the likelihood of a given solution. Although the master equation is somewhat formidable, the intuition is very simple; $p(\boldsymbol{M}|\boldsymbol{\omega}, \boldsymbol{X})$ penalizes Laika whenever it places an object into an unlikely class, and ensuring that the solution reflects the particular memory image, while $p(\boldsymbol{\omega})$ enforces Occam's razor by penalizing Laika whenever it creates an additional class, thus causing it to prefer simpler solutions.

2.5 Typed Pointers

While the simple pointer/integer classification system already produces reasonable results, a key optimization is the introduction of typed pointers. If all of the instances of a class have a pointer at a certain offset, it is probable that the targets of those pointers are also in the same class. As the clustering proceeds and the algorithm becomes more confident of the correct clustering, it changes the address blocks to typed address blocks based on the class of their target. Typed pointers are especially important for small objects, because the class is smaller

atomic type	A machine level type, like a pointer.
block type	Value of an atomic type
data structure	"struct 1". Compound type.
$\boldsymbol{X}_i$	instantiation / object i
i	index of objects
$\boldsymbol{\omega}_j$	class / compound data type j
j	index of classes
k	block offset within a class /object
$\boldsymbol{M}$	the memory image of our process
l	index with a memory image

Table 1: Terms and symbols

and inherently less descriptive, and objects that contain no pointers, which can sometimes be accurately grouped solely by their references. Since it is impossible to measure the prior and posterior probabilities for the classes and pointers of an unknown program, we simply measured the probability that a typed pointer referred to a correctly typed address or an incorrectly typed address. Typed pointers greatly increase the computational complexity of the equation, because the classification of individual objects is no longer independent if one contains a pointer to the other. Worse, when Laika makes a mistake, the typed pointers will cause this error to propagate. Since typed pointer mismatches are weighted very heavily in the master equation, Laika may split the classes that reference the poorly classified data structure as well.

2.6 Dynamically-Sized Arrays

The second small speed bump is the dynamically-sized array. In standard classification, all elements in a class have feature vectors of the same size. Obviously this is not true with data structures, with the most obvious and important being the ubiquitous C string. We handle a dynamic array by allowing objects to "wrap around" modulo the size of a class. In other words, we allow an object to be classified as a contiguous set of instantiations of a given class - an array.

3 Implementation

We implemented Laika in Lisp; the program and its testing tools total about 5000 lines, including whitespace and comments. The program attempts to find good solutions to the master equation when given program images.

Unfortunately our master equation is computationally messy. Usually, unsupervised learning is difficult, while supervised learning is simple: each item is compared against each class and placed in the class that gives the least error. But with our model, if $\boldsymbol{X}_1$ contains a pointer to $\boldsymbol{X}_2$, the type of $\boldsymbol{X}_2$ will affect the block type and therefore the classification error of $\boldsymbol{X}_1$, so even an exact supervised solution is difficult. Therefore our only choice is to rely on an approximation scheme. Laika computes $p(\boldsymbol{\omega}, \boldsymbol{X}|\boldsymbol{M})$ incrementally and uses heuristics to decide which classification changes (e.g., move $\boldsymbol{X}_{233}$ from $\boldsymbol{\omega}_{17}$ to $\boldsymbol{\omega}_{63}$). We leverage typed pointers to compute reasonable changes. For example, whenever an object is added to a class that contains a typed pointer, it tries to move the pointer targets of that object into the appropriate class.

4 Data Structure Detection

Measuring Laika's ability to successfully identify data structures proved surprisingly difficult. Because the programs we measured are not typesafe, there is no way to determine with perfect accuracy the types of individual bytes. The problems begin with unions, continue with strange pointer accesses, and climax with bugs and buffer overflows. Fortunately these are not too common, and we believe our ground truth results are mostly correct.

We used Gentoo Linux to build a complete set of applications and libraries compiled with debugging symbols and minimal optimizations. We then ran our test programs with a small wrapper for *malloc* which recorded the backtrace of each allocation, and used GDB to obtain the corresponding source lines and guesses at assignment variables at types. Because of the convoluted nature of C programs, we manually checked the results and cleaned up things like macros, typedefs, and parsing errors.

Our model proved mostly correct: less than 1% of pointers are unaligned, and only 1% of integers and 3% of strings point into the heap. About 80% of pointers point to the head of objects. Depressingly, the heap is extremely noisy: on average only 45% of a program's heap address space is occupied by active objects, with the rest being malloc padding and unused or uninitialized chunks. Even more depressingly, only 30% of objects contain a pointer. Since Laika relies on building "pointer fingerprints" to classify objects, this means that the remaining 70% of objects are classed almost entirely by the objects that point to them.

We also encountered a rather disturbing number of poor software engineering practices. Several key X Window data structures, as well as the Perl Compatible Regular Expressions library used by Privoxy, use the dreaded *tail accumulator array*. This archaic programming practice appends a dynamically sized array to a fixed structure; the last element is an array of size 0 and each call to malloc is padded with the size of the array. Although this saves a call to malloc, it makes the software much harder to maintain. This hampers our results on all of the X Window applications, because Laika assumes that

Name	Objects	Classes	$p(real\|laika)$	$p(laika\|real)$	$p_{obj}(real\|laika)$	$p_{obj}(laika\|real)$
blackhack	215	6	0.87	1.00	0.87	1.00
xeyes	680	17	0.66	0.68	0.74	0.93
ctorrent	295	19	0.61	0.67	0.60	0.70
privoxy	3881	32	0.90	0.71	0.93	0.82
xclock	2422	54	0.62	0.44	0.72	0.38
xpdf	16846	180	0.61	0.57	0.64	0.56
xarchiver	20993	315	0.52	0.49	0.60	0.60
Average	**6476**	**89**	**0.68**	**0.65**	**0.73**	**0.71**
blackhack-wm	201	8	0.96	1.00	0.96	1.00
ctorrent-wm	249	13	0.80	0.66	0.78	0.73
xeyes-wm	526	22	0.83	0.67	0.79	0.95
privoxy-wm	3615	32	0.92	0.71	0.90	0.88
xclock-wm	2197	43	0.72	0.58	0.79	0.56
xarchiver-wm	7501	89	0.77	0.62	0.80	0.66
xpdf-wm	12995	194	0.63	0.62	0.69	0.64
Average-wm	**3898**	**57**	**0.80**	**0.70**	**0.82**	**0.77**

Table 2: Data Structure Detection Accuracy. The first part of the table shows Laika's accuracy using only the memory image; the second part using the memory image and a list of the sizes and locations of objects. $p_{obj}(real|laika)$ and $p_{obj}(laika|real)$ are the accuracy when known atomic types like *int* or *char* are ignored.

all data structures have a fixed length. To express our appreciation, we sent the X Window developers a dirty sock.

We concentrated on the classification accuracy, the chance that Laika actually placed objects from the same real classes together, as opposed to the block accuracy, the chance that Laika generated correct compositions for its classes, because it is more relevant to correctly identifying viruses. There are two metrics: the probability that two objects from the same Laika class came from the same real class, $p(real|laika)$, and the probability that two objects from the same real class were grouped together, $p(laika|real)$. It is easy to see that the first metric could be satisfied by placing all elements in their own classes, while the second could be satisfied by placing all elements in the same class. Table 2 summarizes the results.

Laika is reasonably accurate but far from perfect, especially on larger programs. The first source of error is data objects (like strings or int arrays), which are difficult to classify without pointers; even some of the real objects contain no pointers. A more interesting problem arises from the variance in size: some classes contain many more objects than others. Generally speaking, Laika merges classes in order to increase the class prior probability $p(\boldsymbol{\omega})$ and splits them in order to increase the the image posterior probability $p(M|\boldsymbol{\omega}, \boldsymbol{X})$. Because the prior has the same weight as a single object, merging two classes that contain many objects will have a much larger effect on the posterior than the prior. For example, Laika may split a binary tree into an internal node class and a leaf node class, because the first would have two address blocks and the second two zero blocks. If there are some 10 objects, then the penalty for creating the additional class would outweigh the bonus for placing the leaf nodes in a more accurate class, but if there are 100 objects Laika is likely to use two classes.

This data also shows just how much Laika's results improve when the random data, freed chunks, and malloc information are removed from the heap. Not only does this result in the removal of 20% of the most random structures, but it also removes malloc padding, which can be uninitialized. Without size information, Laika can easily estimate the size of an object as eight or more times the correct value when there is no pointer to the successor chunk.

To avoid too much mucking around in Laika's dense output files, we can generate relational graphs of the data structures detected. Figure 2 shows a graph of the types discovered by Laika for the test application *Privoxy* without the aid of location information; Figure 3 shows the correct class relationships. Edges in the graph represent pointers; **s1** will have an edge to **s2** if **s1** contains at least one element of type **s2***. We filtered Figure 2 to remove unlikely classes and classes which had no pointers. We are able to easily identify the all the correct matching classes (shown shaded) from the given graph, as well as a few classes that were incorrectly merged by Laika

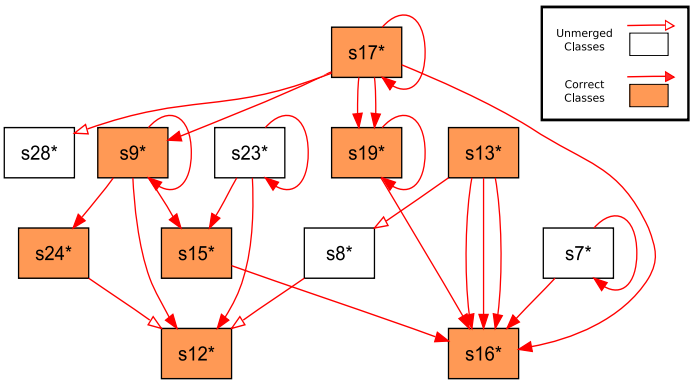


Figure 2: Laika generated type graph for Privoxy

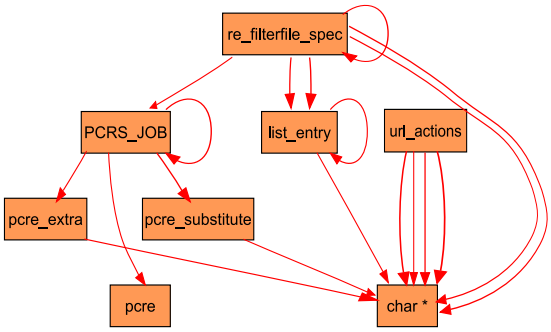


Figure 3: Correct type graph for Privoxy

(shown in white). For instance, type **s17** corresponds directly to the **re_filterfile_spec** structure, whereas types **s9** and **s23** are, in fact, of the same type and should be merged. In addition, by graphing the class relationship in this manner, visually identifying common data structures and patterns is quite simple. For example, a single self-loop denotes the presence of a singly linked list, as shown by the class **s19/list_entry**.

5 Program Detection

Because classifying programs by their data structures already removes all of their code polymorphism, we chose to keep the classifier itself simple. Our virus detector merely runs Laika on the memory images of two programs at the same time and measures how often objects from the different images are placed in the same classes; if many classes contain objects from both programs the programs are likely to be similar. Mathematically, we measured the mixture rate $P(image_i = image_j | class_i = class_j)$ for all object pairs (i, j), which will be closer to 0.5 for similar programs and to 1.0 for dissimilar programs.

Aside from being easy to implement, this approach has several interesting properties. Because Laika is more accurate when given more samples of a class, it is able to discover patterns in the images together that it would miss separately. The mixture ratio also detects changes to the frequency with which the data structures are used. But most importantly, this approach leverages the same object similarity machinery that Laika uses to detect data structures. When Laika makes errors it will tend to make the same errors on the same data, and the mixture ratio focuses on the more numerous - and therefore more accurate - classes, which means that Laika can detect viruses quite accurately even when it does not correctly identify all of their data structures. To focus on the structures that we are more likely to identify correctly, we removed classes that contained no pointers and classes that had very low probability according to the master equation from the mixture ratio. The main disadvantage of the mixture ratio is that the same program can produce different data structures and different ratios when run with different inputs.

We obtained samples of three botnets from Offensive Computing. Agobot/Gaobot is a family of bots based on GPL source released in 2003. The bot is quite object oriented, and because it is open source there are several thousand variants in the wild today. These variants are often fairly dissimilar, as would-be spam lords select different modules, add code, use different compilers and so on. Agobot also contains some simple polymorphic routines. Storm is well known, and its authors have spent considerable effort making it difficult to detect. Kraken, also known as Bobax, has taken off in the spring of 2008. It is designed to be extremely stealthy and, according to some estimates, is considerably larger than Storm [15]. We ran the bots in QEMU to defeat their VM detection and took snapshots of their memory images using WinDbg.

Our biggest hurdle was getting the bots to activate. All of our viruses required direction from a command and control server before they would launch denial of service attacks or send spam emails. For Agobot this was not a problem, as Agobot allocates plenty of data structures on startup. Our Kraken samples were apparently slightly out of date and spent all their time trying to connect to a set of pseudo-random DNS addresses to ask for instructions; most of the data structures we detected are actually allocated by the Windows networking libraries. Our Storm samples did succeed in making contact with the command servers, but this was not an unmixed blessing as their spam emails brought unwanted attention from our obstreperous network administrators. To this day we

Bot	μ_{virus}	σ_{virus}	μ_{other}	σ_{other}	Threshold	Samples	Errors	Est. Accuracy	ClamAV
Agobot	0.64	0.038	0.89	0.053	0.75	19/27	0/0	99.4%	83%
Kraken	0.52	0.021	0.83	0.078	0.58	34/27	0/0	99.8%	85%
Storm	0.51	0.005	0.60	0.015	0.53	20/20	0/0	99.9%	100%

Table 3: Classification Results. For example, we used 17 of our 34 Kraken samples as training data. When the remaining 16 samples were compared against the signature, they produced an average mixture ratio of 0.52 with a standard deviation of 0.021. Of our 27 clean images, we used 13 as training data, and those produced an average mixture ratio with our Kraken sample of 0.83 with a standard deviation of 0.078. The resulting maximum likelihood classifier classifies anything with a mixture ratio of less than 0.58 as Kraken, and has an estimated accuracy of 99.8%; it classified the remaining 17 Kraken samples and 14 standard Windows applications without error. If a new sample was compared with the Kraken signature and produced a mixture rate of 0.56, it would be classified as Kraken, being 1.9σ from the average of the Kraken samples and 3.5σ from the average of the normal samples.

curse their names, especially those who are still alive.

These three botnets represent very different challenges for Laika. Agobot is not merely polymorphic; because it is a source toolkit there are many different versions with considerable differences, while Kraken is a single executable where the differences come almost entirely from the polymorphic engine. Agobot is written in a style reminiscent of MFC, with many classes and allocations on the heap. We think Kraken also has a considerable number of data structures, but in the Kraken images we analyzed the vast majority of the objects were from the Windows networking libraries. This means that the Agobot images were dissimilar to each other owing to the many versions, and also to regular Windows programs because of their large numbers of custom data structures, while the Kraken images were practically identical to each other, but also closer to the regular Windows programs. Neither was Laika's ideal target, which would be a heavily object oriented bot modified only by polymorphic routines. Storm was even worse; by infecting a known process its data structures blended with those of services.exe.

Our virus detector works on each botnet separately; a given program is classified as Kraken or not, then Agobot or not, and so on. We used a simple maximum likelihood classifier, with the single parameter being the mixture ratio between the unknown program and a sample of the virus, which acts as the signature. In such a classifier, each class is represented by a probability distribution; we used a Gaussian distribution as the mixture ratio is an average of the individual class mixture ratios. An unknown sample is placed in the most likely class, i.e. the class whose probability distribution has the greatest value for the mixture ratio of that sample. For a Gaussian distribution, this can be thought of as the class that is closest to the sample, where the distance is normalized by standard deviation.

We used half of our samples as training data to estimate the virus and normal distributions. For normal programs we used 27 standard Windows applications including bittorrent, Skype, Notepad, Internet Explorer, and Firefox. To select the signature from the training set, we tried all of them and chose the one with the lowest predicted error rate. Table 3 summarizes the results. The detector takes from 3 seconds for small applications up to 20 minutes for large applications like Firefox.

Because Storm injects itself into a known process, we had the opportunity to treat it a little differently. We actually used a clean services.exe process as the "virus"; the decision process is reversed and any sample not close enough to the signature is declared to be Storm. This is considerably superior to using a Storm sample, because our Storm images were much less self-similar than the base services.exe images.

5.1 Discussion

The estimated accuracy numbers represent the self-confidence of the model, specifically the overlap of the probability distributions, not its actual tested performance. We included them to give a rough estimate of Laika's performance in lieu of testing several thousand samples. They do not reflect the uncertainty in the estimation of the mean and variance (from some 10-15 samples), which is slightly exacerbated by taking the most discriminatory sample, nor how well the data fit or do not fit our Gaussian model.

It is interesting to note that the accuracy numbers for Kraken and Agobot are roughly comparable despite Agobot containing many unique structures and Kraken using mainly the Windows networking libraries. This occurs because our Kraken samples were extremely similar to one another, allowing Laika could use a very low classification threshold. It is also worth noting that while 99% seems very accurate, a typical computer contains far more than 100 programs.

It would be fairly straightforward to improve our rude 50-line classifier, but even a more complicated version would compare favorably with ClamAV's tens of thousands of lines of code. ClamAV attempts to defeat polymorphic viruses by unpacking and decrypting the hidden executable; this requires a large team of reverse engineers to decipher the various polymorphic methods of thousands of viruses and a corresponding block of code for each.

5.2 Analysis

Since our techniques ignore the code polymorphism of current botnets, it is reasonable to ask whether new "memory polymorphic" viruses will surface if data structure detection becomes common. Because virus detection is theoretically undecidable, such malware is always possible, and the best white hats can do is place as many laborious obstacles as possible in the path of their evil counterparts. We believe that hiding data structures is qualitatively more difficult than fooling signature-based detectors, and in this section we will lay out some counter and counter-counter measures to Laika. Our argument runs in two parts: that high-level structure is harder to obfuscate than lower level structure, and that because high-level structure is so common in programs, we canbe very suspicious of any program that lacks it.

Most of the simplest solutions to obfuscating data structures simply eliminate them. For example, if every byte was XORed with a constant, all of the data structures would disappear. While the classifier would have nothing to report, that negative report would itself be quite damning, although admittedly not all obfuscated programs are malware. Even if the objects themselves were obfuscated, perhaps by appending a large amount of random pointers and integers to each, the classifier would find many objects but no classes, which again would be quite suspicious. Slightly more advanced malware might encrypt half of the memory image, while creating fake data structures from a known good program in the other half. Defeating this might require examining the instruction stream and checking for pointer encryption, i.e. what fraction of pointers are used directly without modification.

To truly fool Laika, a data structure polymorphic virus would need to actually change the layout of its data structures as it spreads. It could do this, perhaps, by writing a compiler that shuffled the order of the fields of all the data structures, and then output code with the new offsets. It is obvious that this kind of polymorphism is much more complicated than the kind of simple instruction insertion engines we see today, requiring a larger payload and increasing the chance that the virus would be enervated by its own bugs. The other option would be to fill the memory image with random data structures and hope that the real program goes undetected in the noise. This increases the memory footprint and reduces the stealthiness of the bot, and has no guarantee of success, since the real data structures are still present. Detecting such viruses would probably require a more complicated descendant of Laika with a more intelligent classifier than the simple mixture ratio.

The greatest advantage of Laika as a virus detector is its orthogonality with existing code-based detectors. At worst, it provides valuable defense in depth by posing malware authors different challenges. At best, it can synergize with code analysis; inspecting the instruction stream may reveal whether a program is obfuscating its data structures.

There are two primary disadvantages to using high-level structure. First, a large class of malware has no high-level structure by the virtue of simplicity. A small kernel rootkit that overwrites a system call table or a bit of shellcode that executes primarily on the stack won't use enough data structures to be detected, and distinguishing a small piece of malware inside a large program like Apache or Linux is more difficult than detecting it in a separate process. However, we believe that there is an important difference between fun and profit. Turning zombie machines into hard currency means putting them to some purpose, be it DoS attacks, spam, serving ads, or other devilry, and that means running some sort of moderately complex process on the infected machines. It is this sort of malware that has been more common lately [1, 12], and it is this sort of malware that we aim to detect.

The second main disadvantage is the substantial increase in resources, in both memory and processor cycles, when compared to current virus scanners. Although a commercial implementation would no doubt be more highly optimized, we do not believe that order of magnitude improvements are likely given the computational complexity of the equations to be solved. Moreover, our data structure detection algorithms apply only to running processes. Worse, those processes will not exercise a reasonable set of their data structures unless they are allowed to perform their malicious actions, so a complete solution must provide a way to blunt any malicious effects prior to detection. This requires either speculatively executing all programs and only committing output after a data structure inspection, or recording all output and rolling back malicious activity. Although we believe that Moore's law will continue to provide more resources, these operations are still not cheap.

5.3 Scaling

Laika runs in linear time in the number of viruses, with moderately high constants. Clearly a commerical version of Laika would require some heuristical shortcuts to avoid stubbornly comparing a test image against some 10^5 virus signature images. The straightforward approach would be to run Laika on the unknown program, record the data structures, compute a signature based on those data structures, use that signature to look up a small set of similar programs in a database, and verify the results with the mixture ratio.

It turns out this is not completely straightforward after all. Consider the most natural approach. The list of data structures may be canonicalized by assigning each structure a unique id. Typed pointer issues can be avoided by assigning a structure a pointer only after canonicalizing the structures to which it points. A program image could then be represented as a vector of structure counts and confidences, and the full mixture ratio computed only for the k nearest neighbors under some distance metric. This approach is extremely brittle. Imagine we have a data structure where one field is changed from 'pointer' to 'zero'. This not only changes the id of that data structure, but also all structures that point to it. In addition, the vector would lie in the space of all known data structures, which would make it hundreds of thousands of elements long. We believe that these difficulties could be solved by a bit of clever engineering, but we have not actually tried to do so.

6 Related Work

Most of the work on the semantic gap so far has come from the security community, which is interested in detecting viruses and determining program behavior by "looking up" from the operating system or VMM towards the actual applications. While most of these problems are either extremely computationally difficult or undecidable in theory, there are techniques that work in practice.

6.1 The Unobfuscated Semantic Gap

Recently, many researchers have proposed running operating system services in separate virtual machines to increase security and reliability. These separated services must now confront the semantic gap between the raw bytes they see and the high-level primitives of the guest OS, and most exploit the fixed interfaces of the processor and operating system to obtain a higher level view, a technique known as virtual machine introspection. Antfarm [18] monitors the page directory register (CR3 in x86) to infer process switches and counts, while Geiger [19] monitors the the page table and through it can make inferences about the guest OS's buffer cache. Intrusion detectors benefit greatly from the protective isolation of a virtual machine [6, 13]; most rely on prior information on some combination of the OS data structures, filesystem format, and processor features.

6.2 The Obfuscated Semantic Gap

Randomization and obfuscation have been used by both attackers and defenders to make bridging the semantic gap more difficult. Address space randomization [4], now implemented in the Linux kernel, randomizes the location of functions and variables; buffer overflows no longer have deterministic side effects and usually cause the program to crash rather than exhibit the desired malicious behavior.

On the other side, virus writers attempt to obfuscate their programs to make them more difficult to disassemble [25]. Compiler-like code transformations such as nop or jump insertion, reording, or instruction substitution [21] are relatively straightforward to implement and theoretically extremely effective: universal virus classification is undecidable [9] and even detecting whether a program is an obfuscated instance of a polymorphic virus is NP-Complete [7]. Even when the virus writer does not explicitly attempt to obfuscate the program, new versions of existing viruses may prove effectively polymorphic [16].

6.3 The Deobfuscated Semantic Gap

Polymorphic virus detectors usually fall into two classes: code detectors and behavior detectors. For example, Christodorescu *et. al* [8] attempt to detect polymorphic viruses by defining patterns of instructions found in a polymorphic worm and searching for them in the unknown binary. Unlike a simple signature checker, these patterns are fairly general and their virus scanner uses a combination of nop libraries, theorem proving and randomized execution. In the end it is capable of detecting instruction (but not memory) reordering, register renaming, and garbage insertion. A recent survey by Singh and Lakhotia [30] gives a good summary of this type of classifier. In addition, Ma *et. al* [26] attempt to classify families of code injection exploits. They use emulated execution to decode shellcode samples and cluster results on executed instruction edit distance. Their results show success in classifying small shellcode samples, but they rely entirely on prior knowledge of specific vulnerabilities to locate and extract these samples.

The second class of detectors concentrate on behavior rather than fingerprinting. These methods usually have either a group of heuristics for malicious behavior [24]

or statistical thresholds [11]. Often they concentrate on semantically higher level features, like system calls or registry accesses rather than individual instructions. Because they are not specific to any particular virus, they can detect unknown viruses, but they often suffer from false positives since benign executables can be very similar to malicious ones.

6.4 Shape Analysis

Modern compilers spend a great deal of time trying to untangle the complicated web of pointers in C programs [32]. Many optimizations cannot be performed if two different pointers in fact refer to the same data structure, and answering this question for structures like trees or hash tables can be difficult. Shape analysis [10, 14] attempts to determine the high-level structure of a program: does a set of allocations form a list, binary tree, or other abstract data type? Although it can enable greater parallelism, shape analysis is very expensive on all but the most trivial programs. Our work attacks the problem of high-level structure from a different angle; although we do not have the source code of the target program, our task is simplified by considering only one memory image.

6.5 Reverse Engineering

Reverse engineering is the art of acquiring human meaning from system implementation. However, most of the work in this field is concentrated on building tools to aid humans discover structure from systems [17, 33], rather than using the information directly. Furthermore, a large amount of the reverse engineering literature [27, 34] is concerned with reverse engineering structure from source code to provide developers with high-level understanding of large software projects.

A more superficially similar technique is Value Set Analysis [3]. VSA can be thought of as pointer aliasing analysis for binary code; it tries to determine possible values for registers and memory at various points in the computation. It is especially useful for analyzing malware. Laika differs from VSA in that it is dynamic rather than static, and that Laika's output is directly used to identify viruses rather than aid reverse engineers.

7 Conclusions

In this paper we have discussed the design and implementation of Laika, a system that detects the data structures of a process given a memory image. The data structures generated by Laika proved surprisingly effective for virus detection. The vast majority of current polymorphic virus detectors work by generating low level fingerprints, but these fingerprints are easily obfuscated by malware writers. By moving the fingerprint to a higher level of abstraction, we increase the difficulty of obfuscation. Laika also provides valuable synnergy to existing code signature based detectors.

Laika exploits the common humanity of programmers; even very flexible fixed interfaces like Von Neumann machines are often used in standard ways. Because fixed interfaces dominate the landscape in systems - often the interface *is* the system - there should be many other such opportunities.

8 Acknowledgments

We would like to thank our shepherd, Geoffrey Voelker, and the anonymous reviewers for their insightful comments and suggestions. We would also like to thank Deepak Ramachandran and Eyal Amir for their help navigating the byzantine labyrinth of machine learning, and Craig Zilles, Matt Hicks, Lin Tan, Shuo Tang, and Chris Grier for reviewing early drafts of this paper. This work was funded by a grant from the Internet Services Research Center (ISRC) of Microsoft Research, and by NSF grant CT-0716768.

References

[1] 2007 malware report: The economic impact of viruses, spyware, adware, botnets, and other malicious code. http://www.computereconomics.com/page.cfm?name=Malware%20Report.

[2] Clamav website. http://www.clamav.org.

[3] BALAKRISHNAN, G., AND REPS, T. Analyzing memory accesses in x86 executables. In *In Comp. Construct* (2004), Springer-Verlag, pp. 5–23.

[4] BHATKAR, S., SEKAR, R., AND DUVARNEY, D. C. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium* (Aug. 2005), USENIX.

[5] BRUSCHI, D., MARIGNONI, L., AND MONGA, M. Detecting self-mutating malware using control flow graph matching. Technical Report, Universitaa degli Studi di Milano, http://idea.sec.dico.unimi.it/ lorenzo/rt0906.pdf.

[6] CHEN, P. M., AND NOBLE, B. D. When virtual is better than real. In *HotOS* (2001), IEEE Computer Society, pp. 133–138.

[7] CHESS, D. M., AND WHITE, S. R. An undetectable computer virus. In *Proceedings of the 2000 Virus Bulletin Conference* (2000).

[8] CHRISTODORESCU, M., JHA, S., SESHIA, S. A., SONG, D., AND BRYANT, R. E. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland 2005)* (Oakland, CA, USA, may 2005), pp. 32–46.

[9] COHEN, F. Computer viruses: Theory and experiments. In *Computers and Security* (1987), pp. 22–35.

[10] CORBETT, J. C. Using shape analysis to reduce finite-state models of concurrent java programs. *ACM Trans. Softw. Eng. Methodol. 9*, 1 (2000), 51–93.

[11] DENNING, D. E. An intrusion-detection model. *IEEE Trans. Softw. Eng. 13*, 2 (February 1987), 222–232.

[12] FRANKLIN, J., PAXSON, V., PERRIG, A., AND SAVAGE, S. An inquiry into the nature and causes of the wealth of internet miscreants. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), ACM, pp. 375–388.

[13] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *NDSS* (2003), The Internet Society.

[14] GHIYA, R., AND HENDREN, L. J. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1996), ACM, pp. 1–15.

[15] GOODIN, D. Move over storm - there's a bigger, stealthier botnet in town. http://www.theregister.co.uk/2008/04/07/kraken_botnet_menace/.

[16] GORDON, J. Lessons from virus developers: The beagle worm history through april 24, 2005. In *SecurityFocus Guest Feature Forum* (2004).

[17] HSI, I., POTTS, C., AND MOORE, M. Ontological excavation: Unearthing the core concepts of applications. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)* (2003).

[18] JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Antfarm: Tracking processes in a virtual machine environment. In *Proceedings of the 2006 USENIX Annual Technical Conference: May 30–June 3, 2006, Boston, MA, USA* (2006).

[19] JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Geiger: monitoring the buffer cache in a virtual machine environment. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2006), ACM Press, pp. 14–24.

[20] JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Vmm-based hidden process detection and identification using lycosid. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (New York, NY, USA, 2008), ACM, pp. 91–100.

[21] JORDAN, M. Dealing with metamorphism, October 2002.

[22] JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 2005 Symposium on Operating Systems Principles (SOSP)* (October 2005), pp. 91–104.

[23] KRUEGEL, C., KIRDA, E., MUTZ, D., ROBERTSON, W., AND VIGNA, G. Polymorphic worm detection using structural information of executables, 2005.

[24] KRUEGEL, C., ROBERTSON, W., AND VIGNA, G. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (Tucson, AZ, December 2004), pp. 91–100.

[25] LINN, C., AND DEBRAY, S. Obfuscation of executable code to improve resistance to static disassembly.

[26] MA, J., DUNAGAN, J., WANG, H. J., SAVAGE, S., AND VOELKER, G. M. Finding diversity in remote code injection exploits. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement* (New York, NY, USA, 2006), ACM, pp. 53–64.

[27] MÜLLER, H. A., ORGUN, M. A., TILLEY, S. R., AND UHL, J. S. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice 5(4)* (December 1993), 181–204.

[28] NUGENT, J., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Controlling your place in the file system with gray-box techniques. In *Proceedings of the 2003 USENIX Annual Technical Conference* (June 2003).

[29] PETRONI, N. L., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot–a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 2004 USENIX Security Symposium* (August 2004).

[30] SINGH, P. K., AND LAKHOTIA, A. Analysis and detection of computer viruses and worms: An annotated bibliography. In *ACM SIGPLAN Notices* (2002), pp. 29–35.

[31] SIVATHANU, M., PRABHAKARAN, V., POPOVICI, F., DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Semantically-Smart Disk Systems. In *Proceedings of the Second USENIX Symposium on File and Storage Technologies (FAST '03)* (San Francisco, CA, March 2003), pp. 73–88.

[32] STEENSGAARD, B. Points-to analysis by type inference of programs with structures and union. In *Proceedings of the 6th International Conference on Compiler Construction* (1996), pp. 136–150.

[33] SUTTON, A., AND MALETIC, J. Mappings for accurately reverse engineering uml class models from c++. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE)* (2005).

[34] WONG, K., TILLEY, S. R., MÜLLER, H. A., AND STOREY, M.-A. D. Structural redocumentation: A case study. *IEEE Software 12*, 1 (1995), 46–54.

Finding and Reproducing Heisenbugs in Concurrent Programs

Madanlal Musuvathi, *Microsoft Research*
Shaz Qadeer, *Microsoft Research*
Thomas Ball, *Microsoft Research*
Gerard Basler, *ETH Zurich*
Piramanayagam Arumuga Nainar, *University of Wisconsin, Madison*
Iulian Neamtiu, *University of California, Riverside*

Abstract

Concurrency is pervasive in large systems. Unexpected interference among threads often results in "Heisenbugs" that are extremely difficult to reproduce and eliminate. We have implemented a tool called CHESS for finding and reproducing such bugs. When attached to a program, CHESS takes control of thread scheduling and uses efficient search techniques to drive the program through possible thread interleavings. This systematic exploration of program behavior enables CHESS to quickly uncover bugs that might otherwise have remained hidden for a long time. For each bug, CHESS consistently reproduces an erroneous execution manifesting the bug, thereby making it significantly easier to debug the problem. CHESS scales to large concurrent programs and has found numerous bugs in existing systems that had been tested extensively prior to being tested by CHESS. CHESS has been integrated into the test frameworks of many code bases inside Microsoft and is used by testers on a daily basis.

1 Introduction

Building concurrent systems is hard. Subtle interactions among threads and the timing of asynchronous events can result in concurrency errors that are hard to find, reproduce, and debug. Stories are legend of so-called "Heisenbugs" [18] that occasionally surface in systems that have otherwise been running reliably for months. Slight changes to a program, such as the addition of debugging statements, sometimes drastically reduce the likelihood of erroneous interleavings, adding frustration to the debugging process.

The main contribution of this paper is a new tool called CHESS for systematic and deterministic testing of concurrent programs. When attached to a concurrent program, CHESS takes complete control over the scheduling of threads and asynchronous events, thereby capturing *all* the interleaving nondeterminism in the program. This provides two important benefits. First, if an execution results in an error, CHESS has the capability to reproduce the erroneous thread interleaving. This substantially improves the debugging experience. Second, CHESS uses systematic enumeration techniques [10, 37, 17, 31, 45, 22] to force every run of the program along a different thread interleaving. Such a systematic exploration greatly increases the chances of finding errors in existing tests. More importantly, there is no longer a need to artificially "stress" the system, such as increasing the number of threads, in order to get interleaving coverage — a common and recommended practice in testing concurrent systems. As a result, CHESS can find in simple configurations errors that would otherwise only show up in more complex configurations.

To build a systematic testing tool for real-world concurrent programs, several challenges must be addressed. First, such a tool should avoid perturbing the system under test, and be able to test the code as is. Testers often do not have the luxury of changing code. More importantly, whenever code is changed for the benefit of testing, the deployed bits are not being tested. Similarly, we cannot change the operating system or expect testers to run their programs in a specialized virtual machine. Therefore, testing tools should easily integrate with existing test infrastructure with no modification to the system under test and little modification to the test harness.

Second, a systematic testing tool must accomplish the nontrivial task of capturing and exploring all interleaving nondeterminism. Concurrency is enabled in most systems via complex concurrency APIs. For instance, the Win32 API [30] used by most user-mode Windows programs contains more than 200 threading and synchronization functions, many with different options and parameters. The tool must understand the precise semantics of these functions to capture and explore the nondeterminism inherent in them. Failure to do so may result in lack of reproducibility and the introduction of false

behaviors.

Finally, a systematic testing tool must explore the space of thread interleavings intelligently, as the set of interleavings grows exponentially with the number of threads and the number of steps executed by each thread. To effectively search large state spaces, a testing tool must not only reduce the search space by avoiding redundant search [16] but also prioritize the search towards potentially erroneous interleavings [32].

The CHESS tool addresses the aforementioned challenges with a suite of innovative techniques. The *only* perturbation introduced by CHESS is a thin wrapper layer, introduced with binary instrumentation, between the program under test and the concurrency API (Figure 1). This layer allows CHESS to capture and explore the nondeterminism inherent in the API. We have developed a methodology for writing wrappers that provides enough hooks to CHESS to control the thread scheduling without changing the semantics of the API functions and without modifying the underlying OS, the API implementation, or the system under test. We are also able to map complex synchronization primitives into simpler operations that greatly simplify the process of writing these wrappers. We have validated our methodology by building wrappers for three different platforms—Win32 [30], .NET [29], and Singularity [19].

CHESS uses a variety of techniques to address the state-explosion problem inherent in analyzing concurrent programs. The CHESS scheduler is non-preemptive by default, giving it the ability to execute large bodies of code atomically. Of course, a non-preemptive scheduler will not model the behavior that a real scheduler may preempt a thread at just about any point in its execution. Pragmatically, CHESS explores thread schedules giving priority to schedules with *fewer* preemptions. The intuition behind this search strategy, called *preemption bounding* [32], is that many bugs are exposed in multithreaded programs by a few preemptions occurring in particular places in program execution. To scale to large systems, we improve upon preemption bounding in several important ways. First, in addition to introducing preemptions at calls to synchronization primitives in the concurrency API, we also allow preemptions at accesses to volatile variables that participate in a data race. Second, we provide the ability to control the components to which preemptions are added (and conversely the components which are treated atomically). This is critical for trusted libraries that are known to be thread-safe.

Today, CHESS works on three platforms and has been integrated into the test frameworks of several product teams. CHESS has been used to find numerous bugs, of which more than half were found by Microsoft testers—people other than the authors of this paper. We emphasize this point because there is a huge difference between the robustness and usability of a tool that researchers apply to a code base of their choice and a tool that has been released "into the wild" to be used daily by testers. CHESS has made this transition successfully. We have reproduced all stress test crashes reported to us so far; as an additional benefit, these crashes were reproduced in small configurations, thereby greatly reducing the debugging effort. Furthermore, we have demonstrated that CHESS scales to large code bases. It has been applied to Dryad, a distributed execution engine for coarse-grained data-parallel applications [21] and Singularity, a research operating system [19].

To summarize, the main contributions of this paper are the following:

- a tool for taking control of scheduling through API-level shimming, allowing the systematic testing of concurrent programs with minimal perturbation;
- techniques for systematic exploration of systems code for fine-grained concurrency with shared memory and multithreading;
- validation of the tool and its architecture and wrapper methodology on three different platforms;
- demonstrating that the tool can find a substantial number of previously unknown bugs, even in well-tested systems;
- the ability to consistently reproduce crashing bugs with unknown cause.

The paper is organized as follows. Section 2 gives an example of how we used CHESS to quickly reproduce a Heisenbug in production code. Section 3 explains the design decisions behind the CHESS scheduler, its basic abstractions, and the wrappers that allow CHESS to effectively control interleaving nondeterminism. Section 4 describes the search strategies that CHESS employs in order to systematically test a concurrent program. Section 5 provides an experimental evaluation of CHESS on several concurrent systems from Microsoft. Finally, section 6 reviews related work.

2 Example

In this section, we describe how we used CHESS to deterministically reproduce a Heisenbug in CCR [8], a .NET library for efficient asynchronous concurrent programming. The code is a client of .NET's System.Threading library, from which it primarily uses monitors, events, and interlocked operations.

The creator of the library reported that a nightly test run had failed. The entire test run consists of many

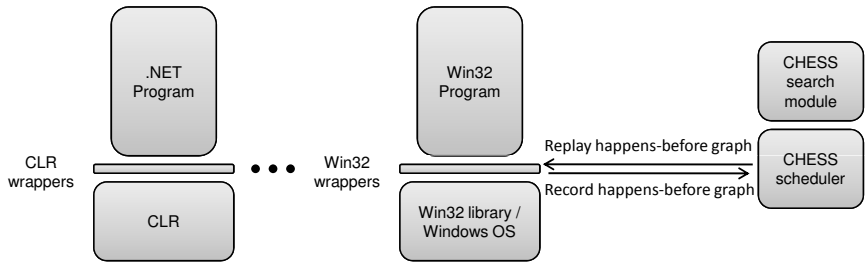


Figure 1: CHESS architecture

smaller unit concurrency tests, each testing some concurrency scenario and each repeatedly executed hundreds to millions of times. The failing test (which did not terminate) previously had not failed for many months, indicating the possibility of a rare race condition.

Our first step was to isolate the offending unit test, which was simply a matter of commenting out the other (passing) tests. The offending test was run several hundred times by the test harness. We changed the harness so it ran the test just once, since CHESS will perform the task of running the test repeatedly anyway. Apart from this change, we did not make any other changes to the test. The entire process took less than thirty minutes, including the time to understand the test framework. As expected, when we ran the modified test repeatedly under the control of the .NET CLR scheduler, it always terminated with success.

We first ran the test under CHESS. In just over twenty *seconds*, CHESS reported a deadlock after exploring 6737 different thread interleavings. It is worth noting that running CHESS a second time produces exactly the same result (deadlock after 6737 interleavings), as the CHESS search strategy is deterministic. However, by simply telling CHESS to use the last recorded interleaving, written out to disk during the execution, CHESS reproduces the deadlock scenario immediately.

We then ran CHESS on the offending schedule under the control of a standard debugger. When CHESS is in control of thread scheduling, it executes the schedule one thread at a time; consequently, single-stepping in the debugger was completely predictable (which is not true in general for a multithreaded program). Furthermore, CHESS provides hooks that allow us to run a thread till it gets preempted. By examining these preemption points, we easily identified the source of the bug, described below.

In this program, the worker threads communicate with each other by exchanging tasks through CCR ports [8], a generic message-passing mechanism in CCR. The bug is due to a race condition in the implementation of a two-phase commit protocol required to atomically process a set of tasks belonging to a port. The bug manifests when a worker A registers to process a set of tasks in a port while a worker B cancels all the tasks in the port. In the erroneous schedule, worker A is preempted after it successfully registers on the port, but before it receives the tasks in the port. Worker B proceeds with the cancellation but gets preempted *after* processing the first task. However, the code fails to set an important field in the port to indicate that cancellation is in progress. As a result, when worker A resumes, it erroneously proceeds to process both tasks, violating the atomicity of the cancellation. This violation results in an exception that leads to a deadlock.

To summarize, with CHESS we were able to reproduce in thirty seconds a Heisenbug that appeared just once in months of testing of a fairly robust concurrency library. This bug requires a specific interleaving between two threads when accessing a port with two or more tasks. Without CHESS, such a complicated concurrency error would have taken several days to weeks to consistently reproduce and debug. It took one of the authors less than an hour to integrate CHESS with the existing test framework, isolate the test that failed from a larger test suite, and deterministically reproduce the Heisenbug. In addition to successfully reproducing many such stress-test crashes, CHESS has also found new concurrency errors in existing, unfailing stress tests.

3 The CHESS scheduler

This section describes the CHESS scheduler and explains how it obtains control over the scheduling of all concurrent activity in a program.

Execution of a concurrent program is highly nondeterministic. We distinguish between two classes of nondeterminism — input and interleaving nondeterminism. The former consists of values provided by the environment that can affect the program execution. For a user-mode program, this consists of return values of all system

calls and the state of memory when the program starts. Interleaving nondeterminism includes the interleaving of threads running concurrently on a multi-processor and the timing of events such as preemptions, asynchronous callbacks, and timers.

The primary goal of the CHESS scheduler is to *capture* all the nondeterminism during a program execution. This allows the scheduler to later reproduce a chosen concurrent execution by replaying these nondeterministic choices. Another equally important goal is to *expose* these nondeterministic choices to a search engine that can systematically enumerate possible executions of a concurrent program. As a concrete example, consider two threads that are blocked on a lock that is currently unavailable. On a subsequent release, the lock implementation is allowed to wake up any of the two blocked threads. Capturing this nondeterminism involves logging which thread was woken up by the system and ensuring that the same happens on a subsequent replay. Exposing this nondeterminism involves notifying the search engine about the two choices available, thereby allowing the engine to explore both possible futures.

Capturing and exposing *all* nondeterministic choices is a nontrivial task. Programs use a large number of concurrency primitives and system calls. The CHESS scheduler needs to understand the semantics of these functions accurately. Failing to capture some nondeterminism can impede CHESS' ability to replay an execution. Similarly, failing to expose nondeterminism can reduce the coverage achieved by CHESS and possibly reduce the number of bugs found. On the other hand, the scheduler has to be careful to not introduce new behaviors that are otherwise not possible in the program. Any such perturbation can result in false error reports that drastically reduce the usability of the tool. Also, the scheduler should not adversely slow the execution of the program. Any slowdown directly affects the number of interleavings CHESS can explore in a given amount of time. Finally, the techniques used to implement the scheduler should allow CHESS to be easily integrated with existing test frameworks. In particular, requiring the program to run under a modified kernel or a specialized virtual machine is not acceptable.

The rest of this section describes how CHESS addresses the aforementioned challenges. We first describe how CHESS handles input nondeterminism in the next subsection and focus on interleaving nondeterminism in following subsections.

3.1 Handling input nondeterminism

Handling input nondeterminism is necessary for deterministic replay. The standard technique for dealing with input nondeterminism involves log and replay [44, 13, 3, 23, 14, 27]. In CHESS, we decided not to implement a complete log and replay mechanism, and shifted the onus of generating deterministic inputs to the user. Most test frameworks already run in a controlled environment, which clean up the state of the memory, the disk, and the network between test runs. We considered the overhead of an elaborate log and replay mechanism unnecessary and expensive in this setting. By using these cleanup functions, CHESS ensures that every run of the program runs from the same initial state.

On the other hand, the scheduler logs and replays input values that are not easily controlled by the user, including functions that read the current time, query the process or thread ids, and generate random numbers. In addition, the scheduler logs and replays any error values returned from system calls. For instance, the scheduler logs the error value returned when a call to read a file fails. However, when the call succeeds CHESS does not log the contents of the file read. While such a log and replay mechanism is easier to implement, it cannot guarantee deterministic replay in all cases. Section 4.2 describes how the search recovers from any unhandled input nondeterminism at the cost of search coverage. Also, CHESS can be easily combined with tools that guarantee deterministic replay of inputs.

3.2 Choosing the right abstraction layer

A program typically uses concurrency primitives provided by different abstraction layers in the system. For Win32 programs, primitives such as locks and threadpools are implemented in a user-mode library. Threading and blocking primitives are implemented in the Windows kernel. In addition, a program can use primitives, such as interlocked operations, that are directly provided by the hardware.

The scheduler is implemented by redirecting calls to concurrency primitives to alternate implementations provided in a "wrapper" library (Figure 1). These alternate implementations need to sufficiently understand the semantics of these primitives in order to capture and expose the nondeterminism in them. For complex primitives, an acceptable design choice is to include an implementation of these primitives as part of the program. For example, if a user-mode library implements locks, the scheduler can treat this library as part of the program under test. Now, the scheduler only needs to understand the simpler system primitives that the lock implementation uses. While this choice makes the scheduler implementation easier, it also prevents the scheduler from exposing all the nondeterminism in the lock implementation. For instance, the library could implement a particular queuing policy that determines the order in which threads acquire locks. Including this implementation as

part of the program prevents the scheduler from emulating other queuing policies that future versions of this library might implement. In the extreme, one can include the entire operating system with the program and implement the scheduler at the virtual machine monitor layer. While this choice might be acceptable for a replay tool like ReVirt [13], it is unacceptable for CHESS because such a scheduler cannot interleave the set of all enabled threads in the program. Rather, it can interleave only those threads that are simultaneously scheduled by the operating system scheduler.

In CHESS, we implement the scheduler for well-documented, standard APIs such as WIN32 and .NET's System.Threading. While these APIs are complex, the work involved in building the scheduler can be reused across many programs.

3.3 The *happens-before* graph

To simplify the scheduler implementation, we abstract the execution of the concurrent program using Lamport's *happens-before* graph [24]. Intuitively, this graph captures the relative execution order of the threads in a concurrent execution. Nodes in this graph represent instructions executed by threads in the system. Edges in the graph form a partial-order that determine the execution order of these instructions. Building the happens-before graph has two important benefits. First, it provides a common framework for reasoning about *all* the different synchronization primitives used by a program. Second, the happens-before graph *abstracts* the timing of instructions in the execution. Two executions that result in the same happens-before graph but otherwise execute at different speeds are behaviorally equivalent. In particular, CHESS can reproduce a particular execution by running the program again on the same inputs and enforcing the same happens-before graph as the original execution.

Each node in the happens-before graph is annotated with a triple—a task, a synchronization variable, and an operation. A task, in most cases, corresponds to the thread executing an instruction. But other schedulable entities, such as threadpool work items, asynchronous callbacks, and timer callbacks are also mapped as tasks. While such entities could possibly execute under the context of the same thread, the CHESS scheduler treats them as logically independent tasks. A synchronization variable represents a resource used by the tasks for synchronizing and communicating with each other, e.g., locks, semaphores, variables accessed using atomic operations, and queues. A fresh synchronization variable is created for a resource the first time it is accessed.

Each resource, depending on its type, allows a set of operations with potentially complex semantics. However, CHESS only needs to understand two bits of information, `isWrite` and `isRelease`, for each of these operations. The bit `isWrite` is used to encode the edges in the happens-before graph of the execution. Intuitively, this bit is true for those operations that change the state of the resource being accessed. If this bit is *true* for a node n, then two sets of happens-before edges are created: (1) edges to n from all preceding nodes labeled with the same synchronization variable as n, and (2) edges from n to all subsequent nodes labeled with the same synchronization variable as n.

The bit `isRelease` is *true* for those operations that unblock tasks waiting on the resource being accessed. The search module in CHESS needs to know the set of enabled tasks at any point in the execution. To maintain this information efficiently, CHESS not only maintains the set of enabled tasks but also the set of tasks waiting for each resource. Upon the execution of an operation whose `isRelease` bit is *true*, CHESS adds these waiting tasks to the set of enabled tasks.

For example, a `CriticalSection` resource provided by WIN32 allows three operations—`EnterCriticalSection`, `ReleaseCriticalSection`, and `TryEnterCriticalSection`. The `isWrite` bit is *true* only for the first two operations and for the last operation whenever it succeeds. The `isRelease` bit is *true* only for `ReleaseCriticalSection`.

Next, we describe how the CHESS scheduler captures the happens-before graph of a concurrent execution. We distinguish between two kinds of inter-thread communication that create edges in the graph. Section 3.4 describes how the scheduler captures communication through synchronization operations, while Section 3.5 describes how the scheduler addresses communication through shared memory.

3.4 Capturing the happens-before graph

During the program execution, CHESS redirects all calls to synchronization operations to a library of wrappers as shown in Figure 1. These wrappers capture sufficient semantics of the concurrency API to provide the abstraction described in Section 3.3 to CHESS. To create this abstraction, the wrappers must: (1) determine whether a task may be disabled by executing a potentially blocking API call, (2) label each call to the API with an appropriate triple of task, synchronization variable, and operation, and (3) inform the CHESS scheduler about the creation and termination of a task. Ideally, in order to minimize perturbation to the system, we would like to meet these three goals without reimplementing the API. To achieve these goals, CHESS maintains state variables for the currently-executing task, the mapping from resource handles to synchronization variables, the set of

enabled threads, and the set of threads waiting on each synchronization variable.

Achieving the first goal requires knowing which API functions can potentially block. We have observed that all such functions invariably have a non-blocking "try" version that indicates the non-availability of the resource using a return value. For example, if the critical section `cs` is not available, then `EnterCriticalSection(cs)` blocks whereas `TryEnterCriticalSection` returns *false*. The wrapper for a blocking function calls the "try" version first, and if this call fails (as indicated by the appropriate return value), then it adds the currently-executing task to the set of tasks waiting on this resource and removes it from the set of enabled tasks. Later, when a release operation is performed on this resource, each task in the set of waiting tasks is moved to the set of enabled tasks.

For the second goal, creating the task and synchronization variable for a particular API call is easily done using the state maintained by CHESS. The first element of the triple is simply the currently-executing task. The second element is obtained by looking up the address of the resource being accessed in the map from resource handles to synchronization variables. Setting the `isWrite` and `isRelease` bits in the third element of the triple requires understanding the semantics of the API call. Note that this understanding does not have to be precise; when in doubt it is always correct, at the cost of efficiency, to set either of those bits to *true*. Conservatively setting the `isWrite` bit to *true* only adds extra edges in the happens-before graph of the execution, adversely affecting the state-caching optimization described later (Section 4.4.2). Conservatively setting the `isRelease` bit to true might unnecessarily move all tasks waiting on a resource to the set of enabled tasks. Later, when the CHESS scheduler attempts to schedule these tasks, they will be moved back into the set of waiting tasks, thus creating some wasteful work. This robustness in the design of our scheduler is really important in practice because the exact behavior of an API function is often unclear from its English documentation. Also, this design allows us to refine the wrappers gradually as our understanding of a particular API improves.

For the third goal, it is important to identify the API functions that can create fresh tasks. The most common mechanism for the creation of a new task are functions such as `CreateThread` and `QueueUserWorkItem`, each of which takes a closure as input. While the former creates a new thread to execute the closure, the latter queues the closure to a threadpool which might potentially multiplex many closures onto the same thread. The CHESS wrappers for both of them are identical. The wrapper first informs CHESS that a new task is being created, creates another closure wrapping the input closure, and then calls the real API function on the new closure. The new closure simply brackets the old closure with calls to the CHESS scheduler indicating the beginning and the end of the task.

A task may be created when a timer is created using `CreateTimerQueueTimer`. Timers are more complicated for several reasons. First, a timer is expected to start after a time period specified in the call to create the timer. The CHESS scheduler abstracts real-time and therefore creates a schedulable task immediately. We feel this is justified because programs written for commodity operating systems usually do not depend for their correctness on real-time guarantees. Second, a timer may be periodic in which case it must execute repeatedly after each expiration of the time interval. Continuing our strategy of abstracting away real-time, we handle periodic timers by converting them into aperiodic timers executing the timer function in a loop. Finally, a timer may be cancelled any time after it has been created. We handle cancellation by introducing a canceled bit per timer. This bit is checked just before the timer function starts executing; the bit is checked once for an aperiodic timer and repeatedly at the beginning of each loop iteration for a periodic timer. If the bit is set, the timer task is terminated.

In addition to the synchronization operations discussed above, CHESS also handles communication primitives involving FIFO queues such as asynchronous procedure calls (APC) and IO completion ports. Each WIN32 thread has a queue of APCs associated with it. A thread can enqueue a closure to the queue of another thread by using the function `QueueUserAPC`. The APCs in the queue of a thread are executed when the thread enters an alertable wait function such as `SleepEx`. Since the operating system guarantees FIFO execution of the APCs, it suffices for the wrappers to pass on the call to the actual function. The treatment of IO completion ports is similar again because the completion packets in the queue associated with an IO completion port are delivered in FIFO order.

Once the wrappers are defined, we use various mechanisms to dynamically intercept calls to the real API functions and forward them to the wrappers. For Win32 programs, we use DLL-shimming techniques to redirect all calls to the synchronization library by overwriting the import address table of the program under test. In addition, we use binary instrumentation to insert a call to the wrapper before instructions that use hardware synchronization mechanisms, such as interlocked operations. For .NET programs, we used an extended CLR profiler [11] that replaces calls to API functions with calls to the wrappers at JIT time. Finally, for the Singularity API, we use a static IL rewriter [1] to make the modifications. Table 1 shows the complexity and the

API	No. of wrappers	LOC
Win32	134	2512
.NET	64	1270
Singularity	37	876

Table 1: Complexity of writing wrappers for the APIs. The LOC does not count the boiler-plate code that is automatically generated from API specifications.

amount of effort required to write the wrappers.

3.5 Capturing data-races by single-threaded execution

Most concurrent programs communicate through shared memory and it is important to capture this communication in the happens-before graph of an execution. If the program is data-race free, then any two conflicting accesses to a shared memory location are ordered by synchronization operations. In this case, the happens-before graph of synchronization operations captured by the wrappers, as described in Section 3.4, is sufficient to order all accesses to shared memory.

Unfortunately, our experience suggests that most concurrent programs contain data-races, many of which are intentional [46]. In this case, the happens-before graph should include additional edges that determine the order of racy accesses. Neglecting these edges may result in inability to replay a given execution. One possible solution is to use a dynamic data-race detection tool [46, 9] that captures the outcome of each data-race at runtime. The main disadvantage of this approach is the performance overhead—current data-race detection slow the execution of the program by an order of magnitude. Therefore, we considered this solution too expensive.

Instead, the CHESS scheduler controls the outcome of data-races indirectly by *enforcing* single-threaded execution [13, 39, 25]. By enabling only one thread at a time, CHESS ensures that two threads cannot concurrently access memory locations. Hence, all data-races occur in the order in which CHESS schedules the threads. While this solves the replay problem, there are two potential downsides of this approach. First, depending on the parallelism in the program, running one thread at a time can slow down the execution of the program. However, this lost performance can be recovered by running multiple CHESS instances in parallel, each exploring a different part of the interleaving state-space. We intend to explore this promising idea in the future. Second, CHESS may not be able to explore both of the possible outcomes of a data-race. This loss of coverage can either result in missed bugs or impact the ability of CHESS to reproduce a Heisenbug that occurs in the wild. We address this problem to an extent, by running the data-race detector on the first few runs of CHESS and then instrumenting the binary with calls to CHESS at the set of data-races found in these runs. This trick has helped us to successfully reproduce all Heisenbugs reported to us (§5), our most important criterian for the success of CHESS.

4 Exploring nondeterminism

The previous section describes how CHESS obtains control at scheduling points before the synchronization operations of the program and how CHESS determines the set of enabled threads at each scheduling point. This section describes how CHESS systematically drives the test along different schedules

4.1 Basic search operation

CHESS repeatedly executes the same test driving each iteration of the test through a different schedule. In each iteration, the scheduler works in three phases: replay, record, and search.

During replay, the scheduler replays a sequence of scheduling choices from a trace file. This trace file is empty in the first iteration, and contains a partial schedule generated by the search phase from the previous iteration. Once replay is done, CHESS switches to the record phase. In this phase, the scheduler behaves as a fair, nonpreemptive scheduler. It schedules a thread till the thread yields the processor either by completing its execution, or blocking on a synchronization operation, or calling a yielding operation such as `sleep()`. On a yield, the scheduler picks the next thread to execute based on priorities that the scheduler maintains to guarantee fairness (§4.3). Also, the scheduler extends the partial schedule in the trace file by recording the thread scheduled at each schedule point together with the set of threads enabled at each point. The latter provides the set of choices that are available but not taken in this test iteration.

When the test terminates, the scheduler switches to the search phase. In this phase, the scheduler uses the enabled information at each schedule point to determine the schedule for the next iteration. Picking the next interesting schedule among the myriad choices available is a challenging problem, and the algorithms in this phase are the most complicated and computationally expensive components of CHESS (§4.4).

The subsequent three subsections describe the key challenges in each of the three phases.

4.2 Dealing with imperfect replay

Unlike stateful model checkers [43, 31] that are able to checkpoint and restore program state, CHESS relies on its ability to replay a test iteration from the beginning to bring a program to particular state. As has been amply demonstrated in previous work [13, 23, 27, 4], perfect replay is impossible without significant engineering and heavy-weight techniques that capture *all* sources of nondeterminism. In CHESS, we have made a conscious decision to not rely on perfect replay capability. Instead, CHESS can robustly handle extraneous nondeterminism in the system, albeit at the cost of the exhaustiveness of the search.

The CHESS scheduler can fail to replay a trace in the following two cases. First, the thread to schedule at a scheduling point is disabled. This happens when a particular resource, such as a lock, was available at this scheduling point in the previous iteration but is currently unavailable. Second, a scheduled thread performs a different sequence of synchronization operations than the one present in the trace. This can happen due to a change in the program control flow resulting from a program state not reset at the end of the previous iteration.

When the scheduler detects such extraneous nondeterminism, the default strategy is to give up replay and immediately switch to the record phase. This ensures that the current test runs to completion. The scheduler then tries to replay the same trace once again, in the hope that the nondeterminism is transient. On a failure, the scheduler continues the search beyond the current trace. This essentially prunes the search space at the point of nondeterminism. To alleviate this loss of coverage, CHESS has special handling for the most common sources of nondeterminism that we encountered in practice.

Lazy-initialization: Almost all systems we have encountered perform some sort of lazy-initialization, where the program initializes a data-structure the first time the structure is accessed. If the initialization performs synchronization operations, CHESS would fail to see these operations in subsequent iterations. To avoid this nondeterminism, CHESS "primes the pump" by running a few iterations of the tests as part of the startup in the hope of initializing all data-structures before the systematic exploration. The downside, of course, is that CHESS loses the capability to interleave the lazy-initialization operations with other threads, potentially missing some bugs.

Interference from environment: The system under test is usually part of a bigger environment that could be concurrently performing computations during a CHESS run. For instance, when we run CHESS on Dryad we bring up the entire Cosmos system (of which Dryad is a part) as part of the startup. While we do expect the tester to provide sufficient isolation between the system under test and its environment, it is impractical to require complete isolation. As a simple example, both Dryad and Cosmos share the same logging module, which uses a lock to protect a shared log buffer. When a Dryad thread calls into the logging module, it could potentially interfere with a Cosmos thread that is currently holding the lock. CHESS handles this as follows. In the replay phase, the interference will result in the current thread being disabled unexpectedly. When this happens, the scheduler simply retries scheduling the current thread a few times before resorting to the default solution mentioned above. If the interference happens in record mode, the scheduler might falsely think that the current thread is disabled when it can actually make progress. In the extreme case, this can result in a false deadlock if no other threads are enabled. To distinguish this from a real deadlock, the CHESS scheduler repeatedly tries scheduling the threads in a deadlock state to ensure that they are indeed unable to make progress.

Nondeterministic calls: The final source of nondeterminism arises from calls to `random()` and `gettimeofday()`, which can return different values at different iterations of the test. We expect the tester to avoid making such calls in the test code. However, such calls might still be present in the system under test that the tester has no control over. We determinize calls to `random()` by simply reseeding the random number generate to a predefined constant at the beginning of each test iteration. On the other hand, we do not determinize time functions such as `gettimeofday()`. Most of the calls to time functions do not affect the control flow of the program. Even when they do, it is to periodically refresh some state in the program. In this case, the default strategy of retrying the execution works well in practice.

4.3 Ensuring starvation-free schedules

Many concurrent programming primitives implicitly assume that the underlying OS schedulers are *strongly fair* [2], that is, no thread is starved forever. For instance, spin-loops are very common in programs. Such loops would not terminate if the scheduler continuously starves the thread that is supposed to set the condition of the loop. Similarly, some threads perform computation until they receive a signal from another thread. An unfair scheduler is not required to eventually schedule the signaling thread.

On such programs, it is essential to restrict systematic enumeration to only fair schedules. Otherwise, a simplistic enumeration strategy will spend a significant amount of time exploring unfair schedules. Moreover, errors found on these interleavings will appear uninteresting to the user as she would consider these interleavings impossible or unlikely in practice. Finally, fair scheduling is

essential because CHESS relies on the termination of the test scenario to bring the system to the initial state. Most tests will not terminate on unfair schedules, and with no other checkpointing capability, CHESS will not be able to bring the system to the initial state.

Of course, it is unreasonable to expect CHESS to enumerate *all* fair schedules. For most programs, there are infinitely many fair interleavings, since any schedule that unrolls a spin-loop an arbitrary but finite number of times is fair. Instead, CHESS makes a pragmatic choice to focus on interleavings that are likely to occur in practice. The fair scheduler, described in detail in [34], gives lower priority to threads that yield the processor, either by calling a yielding function such as `Thread.yield` or by sleeping for a finite time. This immediately restricts the CHESS scheduler to only schedule enabled threads with a higher priority, if any. Under the condition that a thread yields only when it is unable to make progress, this fair scheduler is guaranteed to not miss any safety error [34].

4.4 Tackling state-space explosion

State-space explosion is the bane of model checking. Given a program with n threads that execute k atomic steps in total, it is very easy to show that the number of thread interleavings grows astronomically as n^k. The exponential in k is particularly troublesome. It is normal for realistic tests to perform thousands (if not more) synchronization operations in a single run of the test. To be effective in such large state spaces, it is essential to focus on interesting and potentially bug-yielding interleavings. In the previous section, we described how fairness helps CHESS to focus only on fair schedules. We discuss other key strategies below.

4.4.1 Inserting preemptions prudently

In recent work [32], we showed that bounding the number of preemptions is a very good search strategy when systematically enumerating thread schedules. Given a program with n threads that execute k steps in total, the number of interleavings with c preemptions grows with k^c. Informally, this is because, once the scheduler has picked c out of the k possible places to preempt, the scheduler is forced to schedule the resulting chunks of execution atomically. (See [32] for a more formal argument.) On the other hand, we expect that most concurrency bugs happen [28] because of few preemptions happening at the right places. In our experience with CHESS, we have been able to reproduce very serious bugs using just two preemptions.

On applying to large systems, however, we found that preemption bounding alone was not sufficient to reasonably reduce the size of the state space. To solve this problem, we had to *scope* preemptions to code regions of interest, essentially reducing k. First, we realized that a significant portion of the synchronization operations occur in system functions, such as the C run time. Similarly, many of the programs use underlying base libraries which can be safely assumed to be thread-safe. CHESS does not insert preemptions in these modules, thereby gaining scalability at the cost of missing bugs resulting from adverse interactions between these modules and the rest of the system.

Second, we observed that a large number of the synchronizations are due to accesses to volatile variables. Here we borrow a crucial insight from Bruening and Chapin [5] (see also [41]) — if accesses to a particular volatile variable are always ordered by accesses through other synchronization, then it is not necessary to interleave at these points.

4.4.2 Capturing states

One advantage of stateful model checkers [43, 31] is their ability to cache visited program states. This allows them to avoid exploring the same program state more than once, a huge gain in efficiency. The downside is that precisely capturing the state of a large system is onerous [31, 45]. Avoiding this complication was the main reason for designing CHESS to be stateless.

However, we obtain some advantages of state-caching by observing that we can use the trace used to reach the current state from the initial state as a representation of the current state. Specifically, CHESS maintains for each execution a partially-ordered *happens-before* graph over the set of synchronization operations in the execution. Two executions that generate the same happens-before graph only differ in the order of independent synchronizations operations. Thus, a program that is data-race free will be at the same program state along each of the two executions. By caching the happens-before graphs of visited states, CHESS avoids exploring the same state redundantly. This reduction has the same reduction as a partial-order reduction method called sleep-sets [16] but combines well with preemption bounding [33].

4.5 Monitoring executions

The main goal of CHESS is to systematically drive a concurrent program through possible thread interleavings. Monitoring an interleaving for possible errors, either safety violations or performance problems, is a orthogonal but important problem. Since CHESS executes the program being tested, it can easily catch standard assertion failures such as null dereferences, segmentation faults, and crashes due to memory corruption. The user can also attach other monitors such as memory-leak or

Programs	LOC	max Threads	max Synch.	max Preemp.
PLINQ	23750	8	23930	2
CDS	6243	3	143	2
STM	20176	2	75	4
TPL	24134	8	31200	2
ConcRT	16494	4	486	3
CCR	9305	3	226	2
Dryad	18093	25	4892	2
Singularity	174601	14	167924	1

Table 2: Characteristics of input programs to CHESS

use-after-free detectors [36].

In addition to assertion failures, CHESS also checks each interleaving for deadlocks and livelocks. The wrappers maintain the set of enabled tasks (Section 3.4), allowing CHESS to report a deadlock whenever this set becomes empty. Detecting liveness violations is significantly more difficult and fundamentally requires the fair scheduling capability of CHESS (Section 4.3). Any liveness property is reducible to the problem of fair-termination [42], which is to check whether a program terminates under all fair interleavings. Then, to check if "something good eventually happens", the user writes a test that terminates only when the "good" condition happens. If the program violates this property, the CHESS scheduler will eventually produce a fair interleaving under which the test does not terminate. The user identifies such nonterminating behavior by setting an abnormally high bound on the length of the execution.

We have also implemented monitors for detecting data-races and for detecting whether an execution could result in behavior that is not sequentially consistent [6]. These monitors require trapping all memory accesses and consequently impose a significant runtime overhead. Therefore, we keep these monitors off by default but allow the user to turn them on as needed.

5 Evaluation

In this section, we describe our experience in applying CHESS to several large industry-scale systems.

5.1 Brief description of benchmarks

Table 2 describes the systems on which CHESS has been run on. We briefly describe each of these systems to emphasize the range of systems CHESS is applicable to. Also, the integration of CHESS with the first five systems in Table 2 was done by the users of CHESS, with some help from the authors.

PLINQ [12] is an implementation of the declarative data-parallel extensions to the .NET framework. CDS (Concurrent Data Structures) is a library that implements efficient concurrent data structures. STM is an implementation of software transactional memory inside Microsoft. TPL and ConcRT are two libraries that provide efficient work-stealing implementations of user-level tasks, the former for .NET programs and the latter for C and C++ programs. CCR is the concurrency and coordination runtime [8], which is part of Microsoft Robotics Studio Runtime. Dryad is a distributed execution engine for coarse-grained data-parallel applications [21]. Finally, Singularity [19] is a research operating system.

5.2 Test scenarios

In all these programs, except Singularity, we took existing stress tests and modified them to run with CHESS. Most of the stress tests were originally written to create large number of threads. We modified them to run with fewer threads, for two reasons. Due to the systematic exploration of CHESS, one no longer needs a large number of threads to create scheduling variety. Also, CHESS scales much better when there are few threads. We validate this reduction in the next section.

Other modifications were required to "undo" code meant to create scheduling variety. We found that testers pepper the code with random calls to `sleep` and other yielding functions. With CHESS, such tricks are no longer necessary. On the other hand, such calls impede the coverage achieved with CHESS as the scheduler (§4.3) assumes that a yielding thread is not able to make progress, and accordingly assigns it a low scheduling priority. Another common paradigm in the stress tests was to randomly choose between a variety of inputs with probabilities to mimic real executions. In this case we had to refactor the test to concurrently generate the inputs so that CHESS interleaved their processing. These modifications would not be required if system developers and testers were only concerned about creating interesting concurrency scenarios, and relied on a tool like CHESS to create scheduling variety.

Finally, we used CHESS to systematically test the *entire* boot and shutdown sequence of the Singularity operating system [19]. The Singularity OS has the ability to run as a user process on top of the Win32 API. This functionality is essentially provided by a thin software layer that emulates necessary hardware abstractions on the host. This mechanism alone was sufficient to run the entire Singularity OS on CHESS with little modification. The only changes required were to expose certain low-level synchronization primitives at the hardware abstraction layer to CHESS and to call the shutdown imme-

Programs	Total	Failure / Bug		
		Unk/Unk	Kn/Unk	Kn/Kn
PLINQ	1		1	
CDS	1		1	
STM	2			2
TPL	9	9		
ConcRT	4	4		
CCR	2	1	1	
Dryad	7	7		
Singularity	1		1	
Total	27	21	4	2

Table 3: Bugs found with CHESS, classified on whether the failure and the bug were known or unknown.

diately after the boot without forking the login process. These changes involved ˜300 lines in four files.

5.3 Validating CHESS against stress-testing

Many believe that one cannot find concurrency errors with a small number of threads. This belief is a direct consequence of the painful experience people have of their concurrent systems failing under stress. A central hypothesis of CHESS is that errors in complex systems occur due to *complex* interleavings of *simple* scenarios. In this section, we validate this hypothesis. Recent work [28] also suggests a similar hypothesis.

Table 3 shows the bugs that CHESS has found so far on the systems described in Table 2. Table 3 distinguishes between *bugs* and *failures*. A bug is an error in the program and is associated with the specific line(s) in the code containing the error. A failure is a, possibly nondeterministic, manifestation of the bug in a test run, and is associated with the test case that fails. Thus, a single bug can cause multiple failures. Also, as is common with concurrency bugs, a known failure does not necessarily imply a known bug — the failure might be too hard to reproduce and debug.

Table 3 only reports the number of distinct bugs found by CHESS. The number of failures exceeds this number. As an extreme case, the PLINQ bug is a race-condition in a core primitive library that was the root-cause for over 30, apparently unrelated, test failures. CHESS found a total of 27 bugs in all of the programs, of which 25 were previously unknown. Of these 25 bugs, 21 did not manifest in existing stress runs over many months. The other 4 bugs were those with known failures but unknown cause. In these cases, the tester pointed us to existing stress tests that failed occasionally. CHESS was able to reproduce the failure, within minutes in some cases, with less than ten threads and two preemptions. Similarly, CHESS was able to reproduce two failures that the tester had previously (and painfully) debugged on the STM library. So far, CHESS has succeeded in reproducing every stress-test failure reported to us.

```
LiteEvent::Set(){
   state = SIGNALLED;
   if(kevent)
      lock_and_set_kevent();
}

LiteEvent::Wait(){
   if(state == SIGNALLED)
     return;
   alloc_kevent();
   //BUG:kevent can be 0 here
   kevent.wait();
}

LiteEvent::Dispose(){
   lock_and_delete_kevent();
}
```

Figure 2: A race condition exposed when LiteEvents are used with multiple waiters.

5.4 Description of two bugs

In this section, we describe two bugs that CHESS was able to find.

5.4.1 PLINQ bug

CHESS discovered a bug in PLINQ that is due to an incorrect use of `LiteEvents`, a concurrency primitive implemented in the library. A LiteEvent is an optimization over kernel events that does not require a kernel transition in the common case. It was originally designed to work between exactly two threads — one that calls `Set` and one that calls `Wait` followed by a `Dispose`. Figure 2 contains a simplified version of the code. A lock protects the subtle race that occurs between a `Set` that gets preempted right after an update to `state` and `Dispose`. However, the lock does not protect a similar race between a `Wait` and a `Dispose`. This is because the designer did not intend to use LiteEvents with multiple waiters. However, the PLINQ code did not obey this restriction and CHESS promptly reported this error with just one preemption. As with many concurrency bugs, the fix is easy once the bug is identified.

5.4.2 Singularity bug

CHESS is able to check for liveness properties and it checks the following property by default [34]: every

```
Dispatch(id){
  while (true) {
     Platform.Halt();
     // We wake up on any notification.
     // Dispatch only our id

     if ( PendingNotification(id) ) {
         DispatchNotification(id);
         break;
     }
  }
}

Platform.Halt(){
  if(AnyNotification())
    return;
  Sleep();
}
```

Figure 3: Violation of the good Samaritan property. Under certain cases, this loop results in the thread spinning idly till time-slice expiration.

thread either makes progress towards completing its function or yields the processor. This property detected an incorrect spin loops that never yields in the boot process of Singularity. Figure 3 shows the relevant code snippet. A thread spins in a loop till it receives a particular notification from the underlying hardware. If the notification has a different `id` and thus does not match what it is waiting for, the thread retries without dispatching the notification. On first sight, it appears that this loop yields by calling the `Halt` function. However, `Halt` will return without yielding, if *any* notification is pending. The boot thread, thus, needlessly spins in the loop till its time-slice expires, starving other threads, potentially including the one that is responsible for receiving the current notification.

The developers responsible for this code immediately recognized this scenario as a serious bug. In practice, this bug resulted in "sluggish I/O behavior" during the boot process, a behavior that was previously known to occur but very hard to debug. This bug was fixed within a week. The fix involved changing the entire notification dispatch mechanism in the hardware abstraction layer.

6 Related work

The systematic exploration of the behaviors of executable concurrent programs is not a new idea and has previously occurred in the research areas of software testing and model checking. In contrast to this previous work, this paper is the first to demonstrate the applicability of such systematic exploration to large systems with little perturbation to the program, the runtime, and the test infrastructure.

Carver and Tai [7] proposed repeatable deterministic testing by running a program with an input and explicit thread schedule. The idea of systematic generation of thread schedules came later under the rubric of reachability testing [20]. Recent work in this area includes RichTest [26] which performs efficient search using on-the-fly partial-order reduction techniques, and ExitBlock [5] which observes that context switches only need to be introduced at synchronization accesses, an idea we borrow.

In the model checking community, the idea of applying state exploration directly to executing concurrent programs can be traced back to the Verisoft model checker [17], which is similar to the testing approach in that it enumerates thread schedules rather than states. There are a number of other model checkers, such as Java Pathfinder [43], Bogor [38], CMC [31], and MaceMC [22], that attempt to capture and cache the visited states of the program. CHESS is designed to be stateless; hence it is similar in spirit to the work on systematic testing and stateless model checking. What sets CHESS apart from the previous work in this area is its focus on detecting both safety and liveness violations on large multithreaded systems programs. Effective safety and liveness testing of such programs requires novel techniques—preemption bounding, and fair scheduling—absent from the previous work.

ConTest [15] is a lightweight testing tool that attempts to create scheduling variance without resorting to systematic generation of all executions. In contrast, CHESS obtains greater control over thread scheduling to offer higher coverage guarantees and better reproducibility.

The ability to replay a concurrent execution is a fundamental building block for CHESS. The problem of deterministic replay has been well-studied [25, 39, 44, 13, 3, 23, 14]. The goal of CHESS is to not only capture the nondeterminism but also to systematically explore it. Also, to avoid the inherent cost of deterministic replay, we have designed CHESS to robustly handle some nondeterminism at the cost of test coverage.

The work on dynamic data-race detection, e.g., [40, 35, 46], is orthogonal and complementary to the work on systematic enumeration of concurrent behaviors. A tool like CHESS can be used to systematically generate dynamic executions, each of which can then be analyzed by a data-race detector.

7 Conclusions

In this paper, we presented CHESS, a systematic testing tool for finding and reproducing Heisenbugs in concurrent programs. The systematic testing capability

of CHESS is achieved by carefully exposing, controlling, and searching all interleaving nondeterminism in a concurrent system. CHESS works for three different platforms—Win32, .NET, and Singularity. It has been integrated into the test frameworks of many code bases inside Microsoft and is used by testers on a daily basis. CHESS has helped us and many other users find and reproduce numerous concurrency errors in large applications. Our experience with CHESS indicates that such a tool can be extremely valuable for the development and testing of concurrent programs. We believe that all concurrency-enabling platforms should be designed to enable the style of testing espoused by CHESS.

Acknowledgments

We would like to thank Chris Dern, Pooja Nagpal, Rahul Patil, Raghu Simha, Roy Tan, and Susan Wo for being immensely patient first users of CHESS. We would also like to thank Chris Hawblitzel for helping with the integration of CHESS into the Singularity operating system. Finally, we would like to thank Terence Kelly, our shepherd, and our anonymous reviewers for invaluable feedback on the original version of this paper.

References

[1] Abstract IL — http://research.microsoft.com/projects/ilx/absil.aspx.

[2] APT, K. R., FRANCEZ, N., AND KATZ, S. Appraising fairness in languages for distributed programming. In *POPL 87: Principles of Programming Languages* (1987), pp. 189–198.

[3] BHANSALI, S., CHEN, W.-K., DE JONG, S., EDWARDS, A., MURRAY, R., DRINIĆ, M., MIHOČKA, D., AND CHAU, J. Framework for instruction-level tracing and analysis of program executions. In *VEE 06: Virtual Execution Environments* (2006), ACM, pp. 154–163.

[4] BOOTHE, B. Efficient algorithms for bidirectional debugging. In *PLDI 00: Programming Language Design and Implementation* (2000), pp. 299–310.

[5] BRUENING, D., AND CHAPIN, J. Systematic testing of multithreaded Java programs. Tech. Rep. LCS-TM-607, MIT/LCS, 2000.

[6] BUCKHARDT, S., AND MUSUVATHI, M. Effective program verification for relaxed memory models. In *CAV 08: Computer-Aided Verification* (2008), pp. 107–120.

[7] CARVER, R. H., AND TAI, K.-C. Replay and testing for concurrent programs. *IEEE Softw. 8*, 2 (1991), 66–74.

[8] Concurrency and Coordination Runtime — http://msdn.microsoft.com/en-us/library/bb648752.aspx.

[9] CHOI, J.-D., LEE, K., LOGINOV, A., O'CALLAHAN, R., SARKAR, V., AND SRIDHARAN, M. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI 02: Programming Language Design and Implementation* (2002), pp. 258–269.

[10] CLARKE, E., AND EMERSON, E. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs* (1981), LNCS 131, Springer-Verlag, pp. 52–71.

[11] The CLR profiler — http://msdn.microsoft.com/en-us/library/ms979205.aspx.

[12] DUFFY, J. A query language for data parallel programming: invited talk. In *DAMP* (2007), p. 50.

[13] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI 02: Operating Systems Design and Implementation* (2002).

[14] DUNLAP, G. W., LUCCHETTI, D. G., FETTERMAN, M. A., AND CHEN, P. M. Execution replay of multiprocessor virtual machines. In *VEE 08: Virtual Execution Environments* (2008), ACM, pp. 121–130.

[15] EDELSTEIN, O., FARCHI, E., GOLDIN, E., NIR, Y., RATSABY, G., AND UR, S. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience 15*, 3–5 (2003), 485–499.

[16] GODEFROID, P. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. LNCS 1032. Springer-Verlag, 1996.

[17] GODEFROID, P. Model checking for programming languages using Verisoft. In *POPL 97: Principles of Programming Languages* (1997), ACM Press, pp. 174–186.

[18] GRAY, J. Why do computers stop and what can be done about it? In *Büroautomation* (1985), pp. 128–145.

[19] HUNT, G. C., AIKEN, M., FÄHNDRICH, M., HODSON, C. H. O., LARUS, J. R., LEVI, S., STEENSGAARD, B., TARDITI, D., AND WOBBER, T. Sealing OS processes to improve dependability and safety. In *Proceedings of the EuroSys Conference* (2007), pp. 341–354.

[20] HWANG, G., TAI, K., AND HUNAG, T. Reachability testing: An approach to testing concurrent software. *International Journal of Software Engineering and Knowledge Engineering 5*, 4 (1995), 493–510.

[21] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the EuroSys Conference* (2007), pp. 59–72.

[22] KILLIAN, C. E., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI 07: Networked Systems Design and Implementation* (2007), pp. 243–256.

[23] KONURU, R. B., SRINIVASAN, H., AND CHOI, J.-D. Deterministic replay of distributed java applications. In *IPDPS 00: International Parallel and Distributed Processing Symposium* (2000), pp. 219–228.

[24] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[25] LEBLANC, T. J., AND MELLOR-CRUMMEY, J. M. Debugging parallel programs with instant replay. *IEEE Trans. Comput. 36*, 4 (1987), 471–482.

[26] LEI, Y., AND CARVER, R. H. Reachability testing of concurrent programs. *IEEE Trans. Software Eng. 32*, 6 (2006), 382–403.

[27] LIU, X., LIN, W., PAN, A., AND ZHANG, Z. WiDS checker: Combating bugs in distributed systems. In *NSDI 07: Networked Systems Design and Implementation* (2007), pp. 257–270.

[28] LU, S., PARK, S., SEO, E., AND ZHOU, Y. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS 08: Architectural Support for Programming Languages and Operating Systems* (2008).

[29] .NET Framework 3.5—http://msdn.microsoft.com/en-us/library/w0x726c2.aspx.

[30] Windows API reference—http://msdn.microsoft.com/en-us/library/aa383749(vs.85).aspx.

[31] MUSUVATHI, M., PARK, D., CHOU, A., ENGLER, D., AND DILL, D. L. CMC: A pragmatic approach to model checking real code. In *OSDI 02: Operating Systems Design and Implementation* (2002), pp. 75–88.

[32] MUSUVATHI, M., AND QADEER, S. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI 07: Programming Language Design and Implementation* (2007), pp. 446–455.

[33] MUSUVATHI, M., AND QADEER, S. Partial-order reduction for context-bounded state exploration. Tech. Rep. MSR-TR-2007-12, Microsoft Research, 2007.

[34] MUSUVATHI, M., AND QADEER, S. Fair stateless model checking. In *PLDI 08: Programming Language Design and Implementation* (2008).

[35] O'CALLAHAN, R., AND CHOI, J.-D. Hybrid dynamic data race detection. In *PPOPP 03: Principles and Practice of Parallel Programming* (2003), pp. 167–178.

[36] Rational Purify—http://www-01.ibm.com/software/awdtools/purify.

[37] QUEILLE, J., AND SIFAKIS, J. Specification and verification of concurrent systems in CESAR. In *Fifth International Symposium on Programming*, LNCS 137. Springer-Verlag, 1981, pp. 337–351.

[38] ROBBY, DWYER, M., AND HATCLIFF, J. Bogor: An extensible and highly-modular model checking framework. In *FSE 03: Foundations of Software Engineering* (2003), ACM, pp. 267–276.

[39] RUSSINOVICH, M., AND COGSWELL, B. Replay for concurrent non-deterministic shared-memory applications. In *PLDI 96: Programming Language Design and Implementation* (1996), pp. 258–266.

[40] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems 15*, 4 (1997), 391–411.

[41] STOLLER, S. D., AND COHEN, E. Optimistic synchronization-based state-space reduction. In *TACAS 03* (2003), LNCS 2619, Springer-Verlag, pp. 489–504.

[42] VARDI, M. Y. Verification of concurrent programs: The automata-theoretic framework. *Annals of Pure and Applied Logic 51*, 1-2 (1991), 79–98.

[43] VISSER, W., HAVELUND, K., BRAT, G., AND PARK, S. Model checking programs. In *ASE 00: Automated Software Engineering* (2000), pp. 3–12.

[44] XU, M., BODIK, R., AND HILL, M. D. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. *SIGARCH Comput. Archit. News 31*, 2 (2003), 122–135.

[45] YANG, J., TWOHEY, P., ENGLER, D. R., AND MUSUVATHI, M. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems 24*, 4 (2006), 393–423.

[46] YU, Y., RODEHEFFER, T., AND CHEN, W. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP 05: Symposium on Operating Systems Principles* (2005), pp. 221–234.

Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs

Yin Wang [1,2] Terence Kelly [2] Manjunath Kudlur [1] Stéphane Lafortune [1] Scott Mahlke [1]
[1]*EECS Department, University of Michigan* [2]*Hewlett-Packard Laboratories*

Abstract

Deadlock is an increasingly pressing concern as the multicore revolution forces parallel programming upon the average programmer. Existing approaches to deadlock impose onerous burdens on developers, entail high runtime performance overheads, or offer no help for unmodified legacy code. Gadara automates dynamic deadlock avoidance for conventional multithreaded programs. It employs whole-program static analysis to *model* programs, and Discrete Control Theory to synthesize lightweight, decentralized, highly concurrent logic that *controls* them at runtime. Gadara is safe, and can be applied to legacy code with modest programmer effort. Gadara is efficient because it performs expensive deadlock-avoidance computations *offline* rather than online. We have implemented Gadara for C/Pthreads programs. In benchmark tests, Gadara successfully avoids injected deadlock faults, imposes negligible to modest performance overheads (at most 18%), and outperforms a software transactional memory system. Tests on a real application show that Gadara identifies and avoids both previously known and unknown deadlocks while adding performance overheads ranging from negligible to 10%.

1 Introduction

Deadlock remains a perennial scourge of parallel programming, and hardware technology trends threaten to increase its prevalence: The dawning multicore era brings more cores, but not faster cores, in each new processor generation. Performance-conscious developers of all skill levels must therefore parallelize software, and deadlock afflicts even expert code. Furthermore, parallel hardware often exposes latent deadlocks in legacy multithreaded software that ran successfully on uniprocessors. For these reasons, the "deadly embrace" threatens to ensnare an ever wider range of programs, programmers, and users as the multicore era unfolds.

Our work addresses circular-mutex-wait deadlocks in conventional shared-memory multithreaded programs. Although alternative paradigms such as transactional memory and lock-free data structures attract increasing attention, mutexes will remain important in practice for the foreseeable future. One reason is that mutexes are sometimes preferable, e.g., in terms of performance, compatibility with I/O, or maturity of implementations. Another reason is sheer inertia: Enormous investments, unlikely to be abandoned soon, reside in existing lock-based programs and the developers who write them.

Decades of study have yielded several approaches to deadlock, but none is a panacea. Static deadlock prevention via strict global lock-acquisition ordering is straightforward in principle but can be remarkably difficult to apply in practice. Static deadlock detection via program analysis has made impressive strides in recent years [9,11], but spurious warnings can be numerous and the cost of manually repairing *genuine* deadlock bugs remains high. Dynamic deadlock detection may identify the problem too late, when recovery is awkward or impossible; automated rollback and re-execution can help [38], but irrevocable actions such as I/O can preclude rollback. Variants of the Banker's Algorithm [8] provide dynamic deadlock avoidance, but require more resource demand information than is often available and involve expensive runtime calculations.

Fear of deadlock distorts software development and diverts energy from more profitable pursuits, e.g., by intimidating programmers into adopting cautious coarse-grained locking when multicore performance demands deadlock-prone fine-grained locking. Deadlock in lock-based programs is difficult to reason about because locks are not composable: Deadlock-free lock-based software components may interact to deadlock a larger program [44]. Deadlock-freedom is a *global* program property that is difficult to reason about and difficult to coordinate across independently developed software modules. Non-composability therefore undermines the cor-

nerstones of programmer productivity, software modularity and divide-and-conquer problem decomposition. Finally, insidious corner-case deadlocks may lurk even within single modules developed by individual expert programmers [9]; such bugs can be difficult to detect, and repairing them is a costly, manual, time-consuming, and error-prone chore. In addition to preserving the value of legacy code, a good solution to the deadlock problem will improve new code by allowing requirements rather than fear to dictate locking strategy, and by allowing programmers to focus on modular common-case logic rather than fragile global properties and obscure corner cases.

This paper presents Gadara, our approach to automatically enabling multithreaded programs to dynamically avoid circular-mutex-wait deadlocks. It proceeds in four phases: 1) compiler techniques extract a formal *model* from program source code; 2) Discrete Control Theory methods automatically synthesize *control logic* that dynamically avoids deadlocks in the model; 3) *instrumentation* embeds the control logic in the program where it monitors and controls relevant aspects of program execution; 4) run-time control logic compels the program to behave like the *controlled* model, thereby dynamically avoiding deadlocks. (Gadara is the Biblical place where a miraculous cure liberated a possessed man by banishing en masse a legion of demons.)

Gadara intelligently postpones lock acquisition attempts when necessary to ensure that deadlock cannot occur in a worst-case future. Sometimes the net effect is to alter the scheduling of threads onto locks; in other cases, a thread requesting a lock must wait to acquire it *even though the lock is available*. Gadara may thereby impair performance by limiting concurrency; program instrumentation is another potential performance overhead. Gadara strives to meddle as little as possible while guaranteeing deadlock avoidance, and Discrete Control Theory provides a rigorous foundation that helps Gadara avoid unnecessary instrumentation and concurrency reduction. In practice, we find that the runtime performance overhead of Gadara is typically negligible and always modest—at most 18% in all of our experiments. The computational overhead of Gadara's offline phases (modeling, control logic synthesis, and instrumentation) is similarly tolerable—no worse than the time required to build a program from source. Programmers may selectively override Gadara by disabling the avoidance of some potential deadlocks but not others, e.g., to improve performance in cases where they deem deadlocks highly improbable.

Gadara offers numerous benefits. It dynamically avoids both deterministic/repeatable and also nondeterministic deadlocks. It guarantees that *all* circular-mutex-wait deadlocks are eliminated from a program, and does not introduce new deadlocks or other liveness/progress

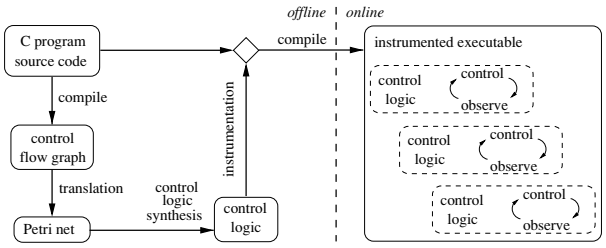


Figure 1: Gadara Architecture.

bugs. It is *safe* and cannot cause a correct program to behave incorrectly. It performs control-synthesis computations offline, greatly reducing the overhead of online control. While it does impose performance overheads, it does not introduce a compulsory global performance bottleneck (e.g., a mandatory "big global lock" or analogous serialization); its control logic is decentralized, fine-grained, and highly concurrent. It works with legacy programs and also with existing *programmers*, requiring no retraining or conceptual reorientation. It neither forbids nor discourages unrestricted I/O. Finally, it relieves programmers of the burden of global reasoning about composability and corner-case deadlock faults.

We have implemented Gadara for C/Pthreads programs. Our experiments show that Gadara enables deadlock-prone software to avoid deadlock at runtime. Gadara furthermore imposes only modest performance overheads, which compare favorably with those of a software transactional memory system. This paper describes the Gadara methodology and our prototype implementation and presents experiments on benchmark software and on a real application, the OpenLDAP directory server. Additional technical details on the Discrete Control Theory techniques underlying Gadara and experiments on randomly generated programs are available in [47].

The remainder of this paper is organized as follows: Section 2 provides an overview of our approach and Section 3 introduces elements of Discrete Control Theory central to Gadara. Section 4 describes how we extract suitable models from program source code, Section 5 explains how Gadara synthesizes control logic from such models, and Section 6 describes Gadara's program instrumentation and run-time control. Section 7 presents our experimental results. Section 8 surveys related work, and Section 9 concludes.

2 Overview of Approach

Figure 1 shows the architecture of Gadara. The offline calculations depicted on the left involve three steps. First, Gadara automatically constructs a formal model from a whole-program Control Flow Graph (CFG) ob-

tained at compile time. This step involves enhancing the standard CFG construction procedure and translating the enhanced graph into a formal model suitable for Discrete Control Theory (DCT) analysis and control logic synthesis. Second, Gadara synthesizes feedback control logic from the model by using DCT techniques. We improve the computational efficiency of standard DCT algorithms by supplementing them with special-purpose strategies that exploit the structure of the model. Third, the synthesized feedback control logic guides source code instrumentation. The key objective of this step is to minimize the online overhead of updating control-related state and implementing the control actions. Finally, online execution of the instrumented program proceeds according to the familiar observation-action paradigm of feedback control. In our problem, control logic delays lock acquisitions to ensure deadlock-free execution of the original program.

An important goal of Gadara's control synthesis phase is a property called "maximally permissive control" (MPC). In the present context, MPC means that the control logic will postpone a lock acquisition only if the program model indicates that deadlock might occur in the future execution of the program if the lock were granted immediately. In other words, control strives to avoid inhibiting concurrency more than necessary to guarantee deadlock avoidance. (One could of course ensure deadlock-freedom in many programs by serializing all threads, but that would defeat the purpose of parallelization.) We are able to make formal statements about MPC because Gadara employs a model-based approach and uses DCT algorithms that guarantee MPC.

Numerous challenges arise in the application of Gadara to real-world programs. We must enhance the standard CFG to obtain a formal model that more accurately captures program behavior; pointer analysis and related difficulties loom large in this area. Imperfections in the formal model can complicate the problem of achieving MPC during the control synthesis phase. Instrumentation must be tolerably lightweight to minimize runtime overhead. Subsequent sections discuss in detail how we address these challenges.

Gadara's limitations fall into two categories: those that are inherent in the problem domain, and those that are artifacts of our current prototype. A trivial example of the former is that Gadara cannot avoid inevitable deadlocks, e.g., due to repeatedly locking a nonrecursive mutex; Gadara issues warnings about such deadlocks. Another limitation inherent to the domain involves the undecidability of general static analysis [23]. It is well known that no method exists for statically determining *with certainty* any non-trivial dynamic/behavioral property of a program, including deadlock susceptibility. However, most real-world programs *do* admit useful static analysis. Gadara builds a program model for which synthesizing deadlock-avoidance control logic is decidable. The model is conservative in the sense that it causes control intervention when static analysis cannot prove that intervention is unnecessary. The net effect is that superfluous control logic sometimes harms performance through instrumentation overhead and concurrency reduction.

A second source of conservatism arises from a limitation in our current prototype: Gadara's offline phases emphasize control flow, performing only limited data-flow analyses; in this respect, Gadara resembles many existing static analysis tools. The code above illustrates the "false paths" problem [9]. Gadara does not currently know that the two conditional branches share identical outcomes if `x` is not modified between them, and therefore mistakenly concludes that this code might acquire the lock but not release it. In the context of a larger program, false paths might cause Gadara to insert superfluous control logic that may needlessly reduce run-time concurrency.

```
if (x)
  lock(L)
...
if (x)
  unlock(L)
```

As with many existing program checkers, imperfect data flow analysis may cause unaided Gadara to identify large numbers of spurious potential deadlocks. We therefore introduce a novel style of programmer-supplied annotation that allows Gadara to eliminate many such "false positives." A first pass of Gadara directs the programmer to problematic functions associated with large numbers of suspected potential deadlocks. The programmer may then annotate these functions to aid a second pass of Gadara, which typically identifies far fewer potential deadlocks by exploiting the annotations. In practice, it is not difficult to annotate real programs correctly and comprehensively. The number of functions that require inspection after the first pass is typically small and it is straightforward for the programmer to annotate them appropriately. Omitted annotations can reduce performance and incorrect annotations can prevent Gadara from avoiding deadlocks already present in the program, but neither compromise safety or correctness. Similarly, illegal pointer casts involving wrapper structures containing mutexes can confuse Gadara and void the guarantee of deadlock-freedom.

Gadara's model-based approach entails both benefits and challenges. Gadara requires that all locking and synchronization be included in its program model; Gadara recognizes standard Pthread functions but, e.g., homebrew synchronization primitives must be annotated. To be fully effective, Gadara must analyze and potentially instrument a whole program. Whole-program analysis can be performed incrementally (e.g., models of library code can accompany libraries to facilitate analysis of client programs), but instrumenting binary-only libraries with control logic would be more difficult. On

the positive side, a strength of a model-based approach is that modeling tends to improve with time, and Gadara's modeling framework facilitates extensions. Petri nets model language features and library functions handled by our current Gadara prototype (calls through function pointers, gotos, libpthread functions) and also extensions (`setjmp()/longjmp()`, IPC). Some phenomena may be difficult to handle well in our framework, e.g., lock acquisition in signal handlers, but most real-world programming practices can be accommodated naturally and conveniently.

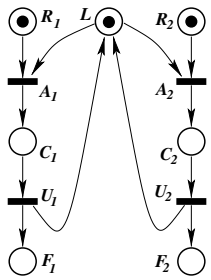
With a modicum of programmer assistance in the form of annotations, Gadara's run-time performance overhead ranges from negligible to modest. Section 7 presents experimental results that quantify these overheads. Before explaining the details of Gadara's phases, we review elements of DCT crucial to Gadara's operation.

3 Discrete Control Theory

Prior research has applied feedback control techniques to computer systems problems [14]. However, this research applied *classical control* to time-driven systems modeled with continuous variables evolving according to differential or difference equations. Classical control cannot model *logical* properties (e.g., deadlock) in event-driven systems. This is the realm of Discrete Control Theory, which considers *discrete event dynamic systems* with discrete state variables and event-driven dynamics. As in classical control, the paradigm of DCT is to synthesize a feedback controller for a dynamic system such that the controlled system will satisfy given specifications. However, the models and specifications that DCT addresses are completely different from those of classical control, as are the modeling formalisms and controller synthesis techniques. DCT is a mature and rigorous body of theory developed since the mid-1980s. This section briefly reviews the specific methods that Gadara employs; see Cassandras & Lafortune for a comprehensive graduate-level introduction to DCT [5].

Finite-state automata and Petri nets [32] are two common modeling formalisms used in DCT, and they are well suited for studying deadlock and other logical correctness properties of discrete event dynamic systems. Given a model of a system in the form of an automaton or a Petri net, DCT techniques can construct feedback controllers that will enforce logical specifications such as avoidance of deadlock, illegal states, and illegal event sequences. DCT is different from (but complementary to) model checking [6] and other formal analysis methods: DCT emphasizes automatically synthesizing a controller that provably achieves given specifications, as opposed to verifying that a given controller (possibly obtained in an ad hoc or heuristic manner) satisfies the specifications. DCT control is correct by construction, obviating the need for a separate verification step.

Wallace et al. [46] proposed the use of DCT in IT automation for scheduling actions in workflow management. We proposed a failure-avoidance system for workflows using DCT [48]. However, these prior efforts assume severely restricted programming paradigms. Gadara moves beyond these limitations and handles multithreaded C programs. DCT has not previously been applied in computer systems for deadlock avoidance in general-purpose software. The finite-automata models of our previous work [48] were adequate since the control flow state spaces of workflows are typically quite small. In the present context, however, automata models do not scale sufficiently for the large C programs that Gadara targets. Gadara therefore employs Petri net models.



As illustrated to the right, Petri nets are bipartite directed graphs containing two types of nodes: *places*, shown as circles, and *transitions*, shown as solid bars. *Tokens* in places are shown as dots, and the number of tokens in each place is the Petri net's state, or *marking*. Transitions model the occurrence of events that change the marking.

Arcs connecting places to a transition represent preconditions of the event associated with the transition. For instance, transition A_1 in our example can occur only if its input places R_1 and L each contain at least one token; in this case, we say that A_1 is *enabled*. Similarly, A_2 is enabled, but all other transitions in the example are disabled. Here, one can think of place L as representing the status of a lock: if L is empty, the lock is not available; if L contains a token, the lock is available. Thus this Petri net models two threads, 1 and 2, that each require the lock. Place R_i represents the request for acquiring the lock for thread i, $i = 1,2$, with transition A_i representing the lock acquisition event. The two lock requests are in conflict: The lock can be granted to only one thread at a time. If transition A_1 *fires*, it consumes one token from each input place R_1 and L and deposits one token in its output place C_1, which models the critical region of thread 1. In general, the firing of a transition consumes tokens from each of its input places and produces tokens in each of its output places; the token count need not remain constant. After A_1 fires, A_2 becomes disabled and must wait for U_1 to occur (lock release by thread 1) before it becomes enabled again. Place F_i represents that thread i has finished.

DCT control logic synthesis techniques for Petri nets exploit the structure of Petri nets for computational efficiency, avoiding an enumeration of the state space (the

set of all markings reachable from a given initial marking) [15]. This is a key advantage of Petri nets over automata, which by construction enumerate the entire state space and thus do not scale to large systems. In a Petri net, state information is distributed and "encoded" as the contents of the places.

Many of the techniques for analyzing the dynamic behavior of a Petri net employ linear algebraic manipulations of matrix representations [32]. In turn, these techniques underlie the control synthesis methodology known as Supervision Based on Place Invariants (SBPI); see [19,31] and references therein. Gadara uses SBPI for control logic synthesis. In SBPI, the control synthesis problem is posed in terms of a set of linear inequalities on the marking of the Petri net. SBPI strategically adds *control places*, and tokens in these, to the Petri net. These control places restrict the behavior of the net and guarantee that the given linear inequalities are satisfied at all reachable markings. Moreover, the control actions provably satisfy the MPC property with respect to the given control specification. In our example Petri net, one could interpret place L as a control place that ensures that the sum of tokens in C_1 and C_2 never exceeds 1. Given this Petri net without place L and its adjacent arcs, and given the constraint that the total number of tokens in C_1 and C_2 cannot exceed 1, SBPI would automatically add L, its arcs, and its initial token. In SBPI, the online control logic is therefore "compiled" offline in the form of the augmented Petri net (with control places). During online execution, the markings of the control places dictate control actions. SBPI terminates with an error message if the system is fundamentally uncontrollable with respect to the given specifications. For Gadara, an example of an uncontrollable program is one that repeatedly acquires a nonrecursive mutex.

Gadara achieves deadlock avoidance by combining SBPI with *siphon* analysis [4]. A siphon is a set of places that never regains a token if it becomes empty. If a Petri net arrives at a marking with an empty siphon, no transition reached by the siphon's places can ever fire. We can therefore establish a straightforward correspondence between deadlocks in a program and empty siphons in its Petri net model.

Gadara employs SBPI to ensure that siphons corresponding to potential circular-mutex-wait deadlocks do not empty. The control places added by SBPI may create new siphons, so Gadara ensures that newly created siphons will never become empty by repeated application of SBPI. Gadara thus resolves deadlocks introduced by its own control logic *offline*, ensuring that no such deadlocks can occur at run time. We have developed strategies for siphon analysis that exploit the special structure of our Petri net models and employ recent results in DCT [27]. These strategies accelerate convergence of Gadara's iterative algorithm while preserving the MPC property.

In summary, siphon analysis and SBPI augment the program's Petri net model with control places that encode feedback control logic; this process does *not* enumerate the reachable markings of the net. The control logic is provably deadlock-free and maximally permissive with respect to the program model. Program instrumentation ensures that the online behavior of the program corresponds to that of the control-augmented Petri net. Gadara control logic and corresponding instrumentation are decentralized, fine-grained, and highly concurrent. Gadara introduces no global runtime performance bottleneck because there is no centralized allocator ("banker") adjudicating lock-acquisition requests, nor is there any global lock-disposition database ("account ledger") requiring serial modification.

4 Modeling Programs

We use the open source compiler OpenIMPACT [36] to construct an augmented control flow graph (CFG) for each function in the input program. Each basic block is augmented with a list of lock variables that are acquired (or released) and the functions that are called within the basic block.

Lock functions We recognize standard Pthreads functions and augment the basic blocks from which they are called. Recognized functions include the mutex, spin, and reader-writer lock/unlock functions and condition variable functions. Large scale software often uses wrapper functions for the primitive Pthread functions. It is beneficial to recognize these wrapper functions, which appear higher up in the call tree where more information is available about the lock involved (e.g., the structures that enclose it). We rely on programmer annotations to recognize wrapper functions. The programmer annotates the wrapper functions at the declaration site using preprocessor directives, along with the argument position that corresponds to the lock variable. Basic blocks that call wrapper functions are marked as acquiring/releasing locks.

Lock variables Every lock function call site in a basic block is also augmented with the lock variable it acquires/releases. Wrapper lock functions typically take *wrapper structures* as arguments, which ultimately embed the lock variable of the primitive type `pthread_mutex_t`. The argument position used in the annotation of a wrapper function automatically marks these wrapper structure types. We define a *lock type* as the type of the wrapper structure that encloses the primitive lock. Basic blocks are augmented with the names of the lock variables if the lock acquisition is directly through the ampersand on a lock variable (e.g.,

`lock(&M)`). If a pointer to a lock type is passed to the lock function at the acquisition site, then the basic block is annotated with the lock type.

Translation To Petri Nets Translating the CFG into a Petri net follows the methodology described in Section 3. A detailed discussion of modeling Pthread functions can be found in [21]. Here, we focus on practical issues related to modeling real-world programs.

We translate each function's CFG into a Petri net in which each transition has a single input place and a single output place. Each basic block in the function is represented by a place in the Petri net, and control transfer from one basic block to another is represented by a transition. Function calls are modeled by substituting into the call site a copy of the callee's Petri net model, i.e., our overall Petri net represents the program's global *inlined* CFG.

Recursion Recursive function calls are handled somewhat like loops when building the inlined CFG for control synthesis. For each function in a recursion, we inline exactly one copy of its Petri net in the model. The recursive call of the function is linked back to the Petri net representing the topmost invocation of the function in the call stack. Control synthesis need not distinguish these "special" loops from normal loops. For control instrumentation, when there are control actions associated with recursive functions, we need to correctly identify entry and return from the recursive call. We augment the function parameter to record the depth of the recursion.

Locks Each statically allocated lock is added to the net as a *mutex place* with one initial token. In addition, every unique lock type (i.e., wrapper structure type) has its own mutex place. An acquisition of a statically allocated lock is modeled as an arc from its corresponding mutex place to the transition corresponding to the lock acquisition. However, an acquisition through a lock pointer is conservatively approximated as an arc from the single place corresponding to the lock type to the corresponding transition. Note that this approximation does not miss any deadlock bugs, but could lead to conservative control. For example, a circular wait detected by Gadara may not be a real deadlock since the threads might be waiting on different lock instances of the same lock type. Section 5 revisits spurious deadlocks and shows how programmer annotations can help Gadara distinguish them.

Thread creation We model thread creation by marking the input places of functions spawned by `pthread_create()` with an infinite number of tokens. This models the scenario in which *any* number of threads could be running concurrently, and deadlock is detected for this scenario. In a real execution, if N is the maximum number of threads that will ever be spawned, and deadlock can occur only when the number of concurrent threads exceeds N, then Gadara will conservatively add control logic to address the spurious deadlock; the runtime cost of this superfluous control is typically a constant. We identify potential thread entry functions in two ways: as statically resolvable pointers passed to `pthread_create()`, and as entry points in the global function call graph; programmer annotations can eliminate some of the latter.

Pruning the Petri net Real programs could result in a large Petri net, slowing offline control logic synthesis. However, logic unrelated to mutexes constitutes the vast majority of real programs. We therefore perform a correctness-preserving performance optimization for the offline control logic synthesis phase by removing such irrelevant areas of program logic from the Petri net model. We prune the original Petri net to a much smaller equivalent by removing functions that do not call lock-related functions directly or indirectly, and then by further reducing the representations of the functions that remain. Function removal involves straightforward analysis of the global function call graph, but function reduction is a more elaborate procedure; see [47] for the details. An important property of our pruning algorithm is that it preserves the mapping from Petri net places to basic blocks in the original program, which facilitates the online control implementation.

5 Offline Control Logic Synthesis

Gadara synthesizes maximally permissive control logic using specialized versions of standard Discrete Control Theory methods. This section explains the basics of our procedures; several correctness-preserving optimizations speed up control logic synthesis, as described in [47].

As explained in Section 3, control logic synthesis in Gadara iteratively identifies siphons in a Petri net corresponding to deadlocks in a real program and uses SBPI to add control places that ensure deadlock avoidance. SBPI operates on a $P \times T$ matrix representation of the Petri net structure, where P and T, respectively, are the number of places and transitions in our pruned Petri net. The computational cost of a single iteration of SBPI is $O(PT^2)$ using naïve methods; Gadara's methods are usually faster because they are specialized to sparse matrices, which are common in practice. In the worst case, the cost of siphon detection is exponential in the number of distinct lock types held by any thread at any instant; better worst-case performance is unlikely because MPC logic synthesis is NP-hard even in our special class of Petri nets [39]. In practice, however, Gadara's entire control logic synthesis phase typically terminates after a single iteration. For a real program like OpenLDAP `slapd`, it is more than an order of magnitude faster than running `make` (seconds vs. minutes).

Deadlock faults may involve distinct lock types, or multiple instances of a single type. Gadara uses standard SBPI control synthesis procedures to identify the former and synthesize satisfactory control logic. Because Gadara's modeling phase substitutes lock *types* for lock *instances*, however, standard DCT techniques detect but cannot remedy deadlock faults involving multiple lock instances of the same lock type. This is not a shortcoming of DCT, but rather a consequence of a modeling simplification forced upon us by the difficulty of data flow analysis, as discussed in Section 4.

Deadlock potentials involving lock instances all of the same type can arise, e.g., in the code on the right. Gadara cannot determine which lock instances are acquired by this loop, nor the acquisition order. Gadara does, however, know that all acquired locks in array `a[]` are of the same lock type (call it `W`). Gadara therefore *serializes the acquisition phases* for locks of this type by adding control logic that prevents more than one thread from acquiring multiple locks of type `W` concurrently, e.g., no more than one thread at a time is permitted to execute the code above.

```
while (...)
  lock(&(a[i]));
```

This approach guarantees deadlock avoidance, but may be deemed unnecessary by programmers: In practice, most real deadlock bugs involve different lock types [9, 28], since it is relatively easy to ensure correct lock ordering within the same lock type. The programmer may therefore choose to disable Gadara's deadlock avoidance for deadlocks involving a single lock type (all such deadlocks, or individual ones).

The control logic that Gadara synthesizes is typically far more subtle than in the simple example discussed above. Most of the subtlety arises from three factors: complicated branching in real programs, the constraint that Gadara's run-time control logic may intervene only by postponing lock acquisitions, and the demand for MPC. We illustrate a more realistic example of deadlock-avoidance control logic using an actual OpenLDAP bug shown in Figure 2, to which we have added clarifying comments; Gadara instrumentation is shown in italics.

Correct lock acquisition order is alphabetical in the notation of the comments. Deadlock occurs if one thread reaches line 10 (holding lock B and requesting A) while another reaches line 2 (holding A, requesting B). Gadara's control logic addresses this fault as follows: Let *t* denote the first thread to reach line 1. Gadara immediately forbids other threads from passing line 1 by postponing this lock acquisition, *even if lock A is available* (e.g., if thread *t* is at line 6). If *t* branches over the body of the `if` on line 7, or if it executes line 13, Gadara knows that *t* cannot be involved in this deadlock bug and therefore permits other threads to acquire the lock at line 1.

```
1 : L_rdwr_wlock(&E.c_rwlock);        /*LOCK(A)*/
    gadara_wlock_and_deplete(&E.c_rwlock,
                             &ctrlplace);
2 : ...
3 : L_mutex_lock(&E.lru_mutex);       /*LOCK(B)*/
4 : ...
5 : L_rdwr_wunlock(&E.c_rwlock);      /*UNLK(A)*/
6 : ...
7 : if (E.c_cursize > E.c_maxsize) {
8 :   ...
9 :   for (elru = E.c_lrutail; elru;
          elru = elprev, i++) {
10:     ...
11:     L_rdwr_wlock(&E.c_rwlock);    /*LOCK(A)*/
12:     ...
13:     L_rdwr_wunlock(&E.c_rwlock); /*UNLK(A)*/
        gadara_replenish(&ctrlplace);
14:     ...
15:   }
16:   ...
17: }
    else gadara_replenish(&ctrlplace);
18: ...
19: L_mutex_unlock(&E.lru_mutex);     /*UNLK(B)*/
```

Figure 2: OpenLDAP deadlock, bug #3494. For clarity, two long strings are abbreviated "`L`" and "`E`."

We instrument the code as follows: we replace the boldface lock-acquisition call on line 1 of the original code with a call to wrapper function `gadara_wlock_and_deplete()`, which atomically depletes the token in the control place that Gadara has added to address this deadlock and calls the program's original lock function. Calls to `gadara_replenish()` restore the token to the control place when it is safe to do so, permitting other threads to pass the modified line 1. MPC guarantees that these replenish calls are inserted as soon as possible, while preserving deadlock-free control. The control place is implemented with a condition variable; the `deplete` function waits on this condition and the `replenish` function signals it.

This example shows that Gadara's control logic is *lightweight*, because it adds only a simple condition variable wait/signal to the code. It is also *decentralized* and therefore *highly concurrent*, because it affects only code that acquires or releases locks A and B; threads acquiring unrelated locks are completely unaffected by the control logic that addresses this deadlock fault, and no central allocator or "banker" is involved. Finally, Gadara's control logic is *fine grained*, because it addresses this specific fault with a dedicated control place; other potential deadlocks are addressed with control places of their own.

Annotations Like many state-of-the-art static analysis tools, Gadara's modeling and control logic synthesis phases do not analyze data flow. As noted in Section 2, false control flow paths lead directly to the detection of

spurious deadlock potentials. Whereas a static analysis tool like RacerX [9] may strive to rank suspected deadlock bugs to aid the human analyst, Gadara is conservative and therefore treats *all* suspected deadlocks equally by synthesizing control logic to dynamically avoid them. Gadara encourages the programmer to add annotations that help rule out spurious deadlocks by showing where annotations are likely to be most helpful; annotations reduce runtime overhead by reducing instrumentation and control.

We found that function-level annotations can greatly reduce the false positive rate with modest programmer effort. Many false positives arise because Gadara believes that a lock type acquired within a function may or may not be held upon return; we call such functions *ambiguous*. Programmer annotations can tell Gadara that a particular lock type is *always* or *never* held upon return from a particular function, thereby disambiguating it. This is a *local* property of the function and is typically easy to reason about. In our experience, a person with little or no knowledge of a large real program such as OpenLDAP can correctly annotate a function in a few minutes. A first pass of Gadara uses lockset analysis [40] to identify ambiguous functions, which are not numerous even in large programs. After the programmer annotates these, Gadara's second pass exploits the annotations to reduce false positives.

We have identified two other patterns, shown on the right, that frequently cause false positives that our annotations can eliminate. In the first case, Gadara cannot tell that the error return occurs only when the lock acquisition fails. In the second case, a function acquires a lock embedded within a dynamically allocated wrapper structure and frees the latter before returning, without bothering to release the enclosed lock. In a variant of the second pattern, the function aborts the entire program without releasing the lock. Annotations reassure Gadara that the enclosing function never returns holding a live mutex.

```
if (OK != lock(&M))
   return ERROR;
...
unlock(&M);
```

```
lock(&S->M);
...
free(&S);
```

6 Instrumentation and Control

The output of the control logic synthesis algorithm is an augmented version of the input Petri net, to which control places with incoming and outgoing arcs to transitions in the original Petri net have been added. An outgoing arc from a control place delays the target transition until a token is available in the control place; the token is consumed when the transition fires. An incoming arc from a transition to a control place replenishes the control place with a token when the transition fires. Outgoing arcs from control places always link to lock acquisition calls, which are the transitions that Gadara's runtime control logic *controls*. Incoming arcs originate at transitions corresponding to lock release calls or branches that the control logic must *observe*.

Gadara's runtime control consists of wrappers for lock-acquisition functions, a control logic state update function, and local variables inserted into the program during instrumentation. The wrappers handle control actions by postponing lock acquisitions; the update function observes selected runtime events and updates control state. Both the wrappers and the update function must correlate program execution state with the corresponding Petri net state. Because we *inlined* functions to create the Petri net, the runtime control logic requires more context than just the currently executing basic block in the function-level CFG. The extra information could be obtained by inspecting the call stack, but we instead instrument functions as necessary with additional parameters that encode the required context. In practice, the control logic usually needs only the innermost two or three functions on the stack, and we add exactly as much instrumentation as required to provide this. For real programs, only a handful of functions require such instrumentation.

As illustrated in Figure 2 and the accompanying discussion, we replace the native lock-acquisition functions with our wrappers only in cases where the corresponding transitions in the Petri net are controlled, i.e., transitions with incoming arcs from a control place. The wrapper function depletes a token from the control place and grants the lock to the thread. If the control place is empty, it waits on a condition variable that implements the control place, which effectively delays the calling thread. For transitions with an outgoing arc to a control place, we insert a control update function that replenishes the token and signals the condition variable of the control place. In certain simple cases, control places can be implemented with mutexes rather than condition variables. In all cases, control is implemented with standard Pthread functions. Gadara carefully orders the locks used in the implementation so that instrumentation itself introduces no deadlocks or other liveness/progress bugs.

The net effect of instrumentation and control is to compel the program's runtime behavior to conform to that of the controlled Petri net model. The control logic intervenes in program execution only by postponing lock acquisition calls. Runtime performance penalties are due to control state update overhead and concurrency reduction from postponing lock acquisitions. The former overhead for a given lock-acquisition function call is proportional to the number of potential deadlocks associated with the call. In practice, we found control update overhead negligible compared to the performance penalty of

postponing lock acquisitions; MPC helps to mitigate the latter.

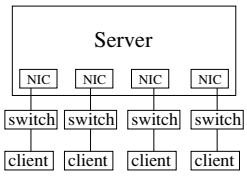
7 Experiments

We conducted experiments to verify Gadara's dynamic deadlock avoidance capabilities, measure its performance overheads, and compare it with an alternative method of guaranteeing deadlock-free execution. Section 7.1 employs several variants of a benchmark application in experiments that exercise Gadara's ability to exorcise injected deadlock bugs, evaluate Gadara's impact on both throughput and response time, and compare Gadara with a software transactional memory (STM) implementation. Section 7.2 shows that Gadara automatically eliminates one known nondeterministic deadlock bug and two previously unreported potential deadlocks in OpenLDAP, and measures Gadara's performance overhead on OpenLDAP. The benchmark deadlock fault involves common-case code, but the OpenLDAP bug involves corner-case code. Section 7.3 briefly summarizes our experience applying Gadara to a deadlock-free program, Apache. Earlier experiments involving randomly generated "dining philosophers" programs are reported in [47].

7.1 Benchmark

We implemented in C/Pthreads a simple client-server publish-subscribe application, PUBSUB, to facilitate fault-injection experiments and comparisons with STM. At a high level, the main logic of the server resembles the "listener pattern" popularized by Miller [29] and Lee [26] to exemplify a simple, useful, and widespread programming pattern that is remarkably troublesome under concurrency. Our PUBSUB server supports three operations: clients may *subscribe* to *channels*, *publish* data to a channel, and request a *snapshot* of all of their current subscriptions.

The server maintains two data structures: a table of each client's subscription lists, indexed by client ID, and a table of channel state and subscriber lists, indexed by channel ID. Both are implemented as open hash tables. Subscribe operations atomically insert a client ID in a channel record and a channel ID in a client's subscription list, thus modifying both tables. Publish operations update the state of a channel and broadcast the result to all of its subscribers. Snapshots first copy the requesting client's list of subscriptions and then traverse the channel table, sending the client the current state of all channels on the list. The server employs a fixed-size pool of worker threads (12 in all of our experiments) and ensures consistent access to shared data via medium-grain locking: one mutex per hash table bucket. An additional mutex per network interface ensures atomicity of snapshot replies. A deadlock-free variant of the server acquires locks in a fixed global order; it is straightforward to inject deadlock faults by perturbing this order. We replaced locks with `atomic { }` blocks to obtain a variant suitable for the Intel C/C++ compiler's prototype STM extension [18, 34].



We ran our benchmark tests in the test environment depicted at right. The server is an HP Compaq dc7800 CMT with 8 GB RAM and a dual-core Intel 2.66 GHz CPU running 64-bit SMP Linux kernel 2.6.22. Four identical dc7800 USD clients with 1 GB RAM and one 2.2 GHz dual-core Intel CPU each running 64-bit SMP Linux kernels 2.6.23 are connected to separate network interface cards on the server via dedicated Cisco 10/100 Mbps Fast Ethernet switches.

Each client machine emulates 1024 clients. Each emulated client first subscribes to 50 different randomly selected channels, then each client machine issues random publish/snapshot requests, with request type, client ID, and channel ID selected with uniform probability; each client machine issues a total of 250,000 requests. The client emulator carefully checks replies for evidence of server-end races, e.g., publication output interleaved with snapshot replies (the latter are supposed to be atomic); we saw no suspicious replies in our tests. The client emulator generates open-arrival requests, which allows us to control server load more readily [41], by using separate threads to issue requests and read replies. We test three variants of the PUBSUB server under two conditions: in heavy-load tests, clients issue requests as rapidly as possible; light-load tests add inter-request delays to throttle request rates to within server capacity.

The table below presents mean server-to-client bandwidths under heavy load and mean response times under light load measured at one of our four symmetric client machines (results on the other client machines are similar). These results are qualitatively representative of a wider range of experiments not reported in detail here.

PUBSUB variant	Heavy Load b/w (Mbit/s)	Light Load resp. time (ms)
DL-free	94.25	10.83
Gadarized	76.88	10.52
STM	47.15	66.70

The deadlock-free variant of PUBSUB (DL-free) represents best-case performance for any deadlock-prone (but race-free) variant. Under heavy load it saturates all four dedicated Fast Ethernet connections to all four client machines, and it serves requests in roughly 11 ms under light load.

Due to the conservatism of Gadara's modeling—specifically, due to the absence of data flow analysis—Gadara cannot distinguish the original deadlock-free PUBSUB from variants containing injected deadlock faults, and Gadara treats both the same way (no annotations were added to PUBSUB, because they would not have helped). The "Gadarized" row in the table therefore represents performance in two scenarios: when Gadara successfully avoids real deadlock bugs, and also when it operates upon a deadlock-free PUBSUB. In the latter case Gadara can only harm performance. In our tests, the harm is moderate: an 18% reduction in throughput under heavy load, and essentially unchanged response times under light load.

The STM results in the last row of the table seem baffling. The optimistic concurrency of TM seems well-suited to the PUBSUB server's data structures and algorithms [25]. PUBSUB-STM should match the performance of the deadlock-free mutex variant under heavy load, and should achieve *faster* response times under light load. The Gadarized variant should (hopefully) perform acceptably, but might reasonably be expected to trail the pack.

The root cause of the TM performance problem lies in the interaction between I/O and the semantics of `atomic { }` blocks. At best, it is very difficult for a TM system to permit concurrency among atomic sections that perform I/O [42, 49]. The Intel STM prototype permits I/O within atomic blocks, but it marks such blocks as "irrevocable" and *serializes* their execution [18]. Like many modern server and client applications [3], PUBSUB performs I/O in critical sections (to ensure that snapshot replies are atomic), and this leads to serialization in the STM variant of PUBSUB.

TM is widely touted as more convenient for the programmer, and less error-prone, than conventional mutexes. Our experience is partly consistent with this view, with several important qualifications. Defining atomic sections is indeed easier than managing locks. Our performance results show, however, that this convenience can carry a price: Mutexes are a more nuanced language for expressing I/O concurrency opportunities than atomic sections, and performance may suffer if the latter are used. If our goal is to exploit available physical resources fully, we would currently choose locks over TM; Gadara removes a major risk associated with this choice. The STM implementation that we used furthermore requires additional work from the programmer beyond defining atomic sections, e.g., function annotations; the total amount of programmer effort required to STM-ify PUBSUB was greater than that of using Gadara. Moreover, some of the extra work requires great care: incorrect STM function annotations can yield *undefined* behavior [18], whereas omitted or incorrect Gadara annotations have less serious consequences.

7.2 OpenLDAP

OpenLDAP is a popular open-source implementation of the Lightweight Directory Access Protocol (LDAP). The OpenLDAP server program, `slapd`, is a high-performance multithreaded network server. We applied Gadara to `slapd` in version 2.2.20, which has a confirmed deadlock bug [37]. The bug was fixed in 2.2.21 but returned in 2.3.13 when new code was added.

The `slapd` program has 1,795 functions, of which 456 remain after pruning. Control flow graph generation and Gadara's modeling phase took roughly as long as a full build of the `slapd` program; two passes of control logic synthesis each took far less time (a few seconds).

In addition to standard Pthreads lock functions, we annotated six pairs of lock and unlock functions that operate upon file or database locks or call Pthreads lock functions through pointers. OpenLDAP contains 41 lock types, i.e., distinct types of wrapper structures that contain locks. After model translation and reduction, the model contains two separate Petri nets that may potentially deadlock, one with two lock types and the other with 15 lock types. The model contains separate Petri nets because different modules of the program use different subsets of locks. We apply Gadara to each separate net independently, which reduces the computational complexity of control logic synthesis without changing the resulting control logic. Gadara's first pass completed in a few seconds and reported 25 ambiguous functions (i.e., the set of locks held on return was ambiguous). We manually inspected these functions and annotated 21; ambiguities in the remaining four functions were genuine. A programmer not deeply familiar with the source code required a little over an hour to disambiguate `slapd`'s functions.

Disambiguation allows Gadara's second pass to construct a more accurate model with fewer false execution paths and fewer spurious deadlock potentials. The second-pass model of `slapd` contains four separate Petri nets that may deadlock, three with two lock types and one with four. Each separate Petri net contains one siphon. It was easy to confirm manually that the known deadlock bug corresponds to one of these siphons. Of the remaining three siphons, one was clearly a false positive; it was a trivial variant of the false paths pattern in Section 2 that spans two functions and that our current prototype does not weed out, even after disambiguation. The last two siphons correspond to genuine deadlock faults. We disabled Gadara control for the obvious false positive and allowed Gadara to address the three genuine faults.

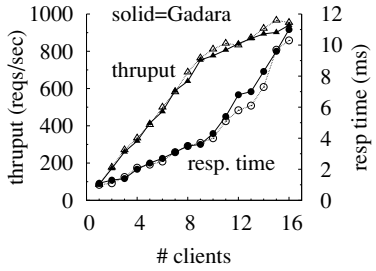


Figure 3: Modify workload.

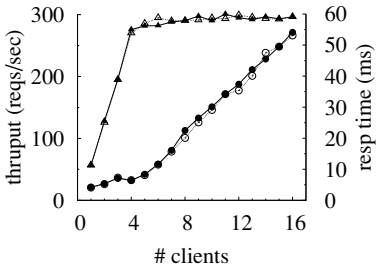


Figure 4: Search workload.

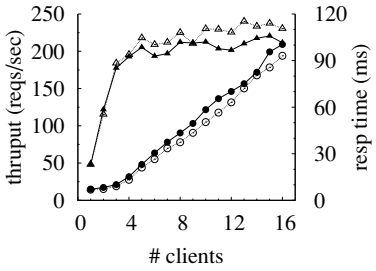


Figure 5: Add/Del workload.

The control synthesis algorithm terminated in a few seconds, after a single iteration.

We first tested whether the Gadarized `slapd` successfully avoids the known deadlock bug, which resides among database cache functions that participate in insertion and eviction operations on the `slapd` application-level cache. The bug is nondeterministic and hard to reproduce, but we were able to reliably trigger it after inserting four `sched_yield()` calls immediately before a thread requests an additional lock while holding a particular lock. We configured `slapd` with a small cache size to trigger frequent cache evictions. After these changes, we were able to reproduce the deadlock bug reliably within one minute or less with a workload consisting of a mixture of add, delete and modify requests. An otherwise-identical Gadarized version of the same `slapd`, however, successfully serves the same workload indefinitely without deadlock or other difficulty.

Our next experiments compare performance between original and Gadarized `slapd` variants (neither containing the `sched_yield()` calls inserted for our deadlock-avoidance test above). Our OpenLDAP clients submit three different workloads to a `slapd` that contains a simulated "employee database" directory: search workloads perform lookups on indexed fields of randomly selected directory entries; modify workloads alter the contents of randomly selected entries by adding new field and deleting a field; and add/delete workloads create and remove randomly generated entries. We vary the number of clients between 1 and 16, and we locate the client emulators on the same server as `slapd` to make it easier to overload the latter.

Preliminary tests showed that Gadara overhead is negligible when `slapd` is configured normally, because performance is disk bound and because the deadlock faults that Gadara addressed involve code paths that execute infrequently. We therefore took extraordinary measures to ensure that `slapd` is not disk bound and that the faulty code of the known bug executes frequently: we used a small directory (100 entries), disabled database synchronization, and configured `slapd` to serve replies from in-memory data via the "`dirtyread`" directive. This configuration is highly atypical but is required to trigger any Gadara overhead at all for the OpenLDAP deadlock bug.

Figures 3, 4, and 5 present average response time and throughput measurements for our three workloads. In terms of both performance metrics, Gadara imposes overheads of 3–10% for the Modify and Add/delete workloads; overhead is negligible for the Search workload. The difference occurs because the Gadara instrumentation and control logic are triggered only in functions that add and delete items from the `slapd` application-level cache. Modify and Add/delete workloads cause cache insertions and deletions, and therefore incur Gadara overhead. The Search workload, however, performs only cache lookups, and therefore avoids Gadara overhead.

7.3 Apache

We applied Gadara to Apache `httpd` version 2.2.8. The program has 2,264 functions and 12 distinct lock types. The first pass of Gadara identifies 28 ambiguous functions. Almost all ambiguities involve error checking in lock/unlock functions (if the attempt to acquire a lock fails, return immediately) so it was easy to disambiguate these functions. After we appropriately annotate them, Gadara reports no circular-mutex-wait deadlock, and therefore Gadara inserts no control logic instrumentation. In Apache, most functions acquire at most one lock and release it before returning. This lock usage pattern is restrictive, but makes it relatively easy to write deadlock-free programs. Gadara's analysis of `httpd` is consistent with the Apache bug database, which reports no circular-mutex-wait deadlocks in any 2.x version of Apache. Two reported deadlocks in the bug database involve inter-process communication, not mutexes [28].

7.4 Discussion

Our experience shows that Gadara handles large real programs, and it is easier to Gadarize a program than migrate it to atomic sections. Experiments with our prototype implementation show that Gadara successfully avoids deadlocks in deadlock-prone programs with little or no adverse impact on performance. As illustrated by

our OpenLDAP results, Gadara works particularly well for corner-case deadlock faults in infrequently-executed code; Gadara eliminates such faults with modest programmer effort and with low performance overhead even under adverse conditions. Our benchmark tests show that the performance overhead may be tolerable even when Gadara corrects deadlock faults in common-case code paths. The history of the OpenLDAP bug furthermore shows that Gadara may be a reasonable alternative to the straightforward approach of manually fixing deadlock faults—the latter was done for the `slapd` deadlock we discuss, but the bug returned many versions later. The cost of running Gadara to eliminate corner-case deadlocks in each new version may compare favorably with the cost of repeated manual repair.

8 Related Work

There are four basic approaches to dealing with deadlock in multithreaded programs that employ locks: static prevention, static detection, dynamic detection, and dynamic avoidance. Static deadlock prevention by acquiring locks in a strict global order is straightforward but rarely easy. Experience has shown that it is cumbersome at best to define and enforce a global lock acquisition order in complex, modular, multi-layered software. Lock ordering can become untenable in software developed by independent teams separated in both time and geography. Indeed, corner-case lock-order bugs arise even in individual modules written by a single expert programmer [9]. Our contribution is to perform systematic global reasoning in the control logic synthesis phase of Gadara, relieving programmers of this burden.

Static detection uses program analysis techniques to identify potential deadlocks. Examples from the research literature include the Extended Static Checker (ESC) [11] and RacerX [9]; commercial tools are also available [43]. Adoption, however, is far from universal because spurious bug reports are common for real-world programs, and it can be difficult to separate the wheat from the chaff. Repair of real defects identified by static analysis remains manual and therefore time-consuming, error-prone, and costly. By contrast, Gadara automatically repairs deadlocks.

Dynamic detection does not suffer from false positives, but by the time deadlock is detected, recovery may be awkward or impossible. Automated rollback and reexecution can eliminate the burden on the programmer and guarantee safety in a wider range of conditions [38], but irrevocable actions such as I/O may preclude rollback. Dynamic detection of *potential* deadlocks (inconsistent lock acquisition ordering) can complement static deadlock detection [1, 2].

Dijkstra's "Banker's Algorithm" dynamically avoids *resource* deadlocks by postponing requests, thereby constraining a set of processes to a safe region from which it is possible for all processes to terminate [7, 13, 22] (mutex deadlocks call for different treatment because, unlike units of resources, mutexes are not fungible). Holt [16,17] improved the efficiency of the original algorithm and introduced a graphical understanding of its operation. While the classic Banker's Algorithm is sometimes used in real-time computing, its usefulness in more general computing is limited because it requires knowledge of a program's dynamic resource consumption that is difficult to specify. The Banker's Algorithm has been applied to manufacturing systems under assumptions and system models inappropriate for our domain [39, 45].

Generalizations of the Banker's Algorithm address mutex deadlocks, and some can exploit (but do not provide) models of program behavior of varying sophistication [10, 12, 24, 30, 33, 50]. Gadara differs in several respects. First, it both generates and exploits models of real programs with greater generality and fidelity. More importantly, Gadara's online computations are much more efficient. In contrast to the Banker's Algorithm's expensive online safety checks, Discrete Control Theory allows Gadara to perform most computation *offline*, greatly reducing the complexity of online control. Finally, Banker-style schemes employ a central allocator whose "account ledger" must be modified whenever resources/locks are allocated. In an implementation, such write updates may be *inherently serial*, regardless of the concurrency control mechanisms that ensure consistent updates (conventional locks, lock-free/wait-free approaches, or transactional memory). For example, in the classic single-resource Banker's Algorithm, updates to the "remaining units" variable are necessarily serial. As a consequence, performance suffers doubly: acquisitions are serialized, and each acquisition requires an expensive safety check. By contrast, Gadara's control logic admits true concurrency because it is decentralized; there is no central controller or global state, and lock acquisitions are not globally serialized.

Nir-Buchbinder et al. describe a two-stage "exhibiting/healing" scheme that prevents previously observed lock discipline violations from causing future deadlocks [35]. The "exhibiting" phase attempts to trigger lock discipline violations during testing by altering lock acquisition timing. "Healing" then addresses the potential deadlocks thus found by adding gate locks to ensure that they cannot cause deadlocks in subsequent executions. The production runtime system detects new lock discipline violations and also deadlocks caused by gates; it recovers from the latter by canceling the gate, and ensures that similar gate-induced deadlocks cannot recur. As time goes on, programs progressively become

deadlock-free as both native and gate-induced deadlocks are healed. The runtime checks of the healing system require time linear in the number of locks currently held and requested; lower overhead is possible if deadlock detection is disabled. Jula & Candea describe a deadlock "immunization" scheme that dynamically detects specific deadlocks, records the contexts in which they occur, and dynamically attempts to avoid recurrences of the same contexts in subsequent executions [20]. This approach dynamically performs lock-acquisition safety checks on an allocation graph; the computational complexity of these checks is linear, polynomial, and exponential in various problem size parameters. Like healing, immunization can introduce deadlocks into a program.

Gadara differs from healing and immunization in several respects: Whereas these recent proposals perform centralized online safety checks involving graph traversals, Gadara's control logic is much less expensive because DCT enables it to perform most computation—including the detection and remediation of avoidance-induced deadlocks—*offline*. Healing and immunity tell the user what deadlocks have been addressed, but not whether any deadlocks remain. By contrast, Gadara guarantees that all deadlocks are eliminated at compile time, ensuring that they never occur in production. Whereas the computational complexity of the safety checks in healing and immunity depend on runtime conditions, in Gadara the dynamic checks associated with a lock acquisition (i.e., the control places incident to a lock acquisition transition) are known *statically*; programmers may therefore choose to manually repair deadlock faults that entail excessive control logic and allow Gadara to address the deadlocks that require little control logic. Whereas healing's guard locks essentially coarsen a program's locking, Gadara's maximally permissive control logic synthesis allows more runtime concurrency. The price Gadara pays for its advantages is the need for whole-program analysis, reliance on programmer annotations to improve performance, and the possibility of performance degradation due to superfluous control logic for spurious deadlock faults detected during offline analysis.

9 Conclusions

To the best of our knowledge, Gadara is the first approach to circular-mutex-wait deadlock that does all of the following: Leverages deep knowledge of applications; safely eliminates all circular-mutex-wait deadlocks; places no major new burdens on programmers; remains compatible with the installed base of compilers, libraries, and runtime systems; imposes modest performance overheads on real programs serving realistic workloads; and liberates programmers from fear of deadlock, empowering them to implement more ambitious locking strategies. In Gadara, compiler technology supplies deep whole-program analysis that yields a global model of all possible program behaviors, including corner-cases likely to evade testing. Discrete Control Theory combines the strengths of offline analysis and control synthesis with online observation and control to dynamically avoid deadlocks in concurrent programs. Thanks to DCT, Gadara's control logic is lightweight, decentralized, fine-grained, and highly concurrent. Gadara exploits the natural synergy between the strengths of DCT and compiler technology to solve one of the most formidable problems of concurrent programming. Gadara is set apart from alternative approaches in that it provides a whole-program model for analyzing and managing concurrency.

10 Acknowledgments

The research of Wang, Lafortune, and Mahlke is supported in part by NSF grants ECCS-0624821, CCF-0819882, and CNS-0615261, and by an HP Labs Open Innovation award. We thank Marcos Aguilera, Eric Anderson, Hans Boehm, Dhruva Chakrabarti, Peter Chen, Pramod Joisha, Xue Liu, Mark Miller, Brian Noble, and Michael Scott for encouragement, feedback, and valuable suggestions. Ali-Reza Adl-Tabatabai answered questions about the Intel STM prototype. We are grateful to Kumar Goswami and Norm Jouppi for funding, to Laura Falk and Krishnan Narayan for IT support, to Kelly Cormier and Cindy Watts for administrative and logistical assistance, and to Shan Lv and Soyeon Park for sharing details concerning [28]. Finally we thank our shepherd, Remzi Arpaci-Dusseau, and the anonymous OSDI reviewers for many helpful suggestions.

References

[1] AGARWAL, R., AND STOLLER, S. D. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *Proc. Workshop on Parallel and Distributed Systems: Testing and Debugging* (2006).

[2] AGARWAL, R., WANG, L., AND STOLLER, S. D. Detecting potential deadlocks with static analysis and runtime monitoring. In *Proc. Parallel and Distributed Systems* (2006), vol. 3875 of *LNCS*, Springer-Verlag.

[3] BAUGH, L., AND ZILLES, C. An analysis of i/o and syscalls in critical sections and their implications for transactional memory. In *TRANSACT* (2007).

[4] BOER, E. R., AND MURATA, T. Generating basis siphons and traps of Petri nets using the sign incidence matrix. *IEEE Trans. on Circuits and Systems—I 41*, 4 (1994).

[5] CASSANDRAS, C. G., AND LAFORTUNE, S. *Introduction to Dsicrete Event Systems*, second ed. Springer, 2007.

[6] CLARKE, E., GRUMBERG, O., AND PELED, D. *Model Checking*. MIT Press, 2002.

[7] DIJKSTRA, E. W. Solution of a problem in concurrent programming control. *CACM 8*, 9 (1965).

[8] DIJKSTRA, E. W. *Selected Writings on Computing*. Springer-Verlag, 1982, ch. The Mathematics Behind the Banker's Algorithm.

[9] ENGLER, D., AND ASHCRAFT, K. RacerX : effective, static detection of race conditions and deadlocks. In *SOSP* (2003).

[10] FINKEL, R., AND MADDURI, H. H. An efficient deadlock avoidance algorithm. *Inf. Process. Lett. 24*, 1 (1987).

[11] FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA:, R. Extended static checking for Java. In *PLDI* (2002).

[12] GOLD, E. M. Deadlock prediction: Easy and difficult cases. *SIAM J. Comput. 7*, 3 (1978).

[13] HABERMANN, A. N. Prevention of system deadlocks. *CACM 12*, 7 (1969).

[14] HELLERSTEIN, J. L., DIAO, Y., PAREKH, S., AND TILBURY, D. M. *Feedback Control of Computing Systems*. Wiley, 2004.

[15] HOLLOWAY, L., KROGH, B., AND GIUA, A. A survey of Petri net methods for controlled discrete event systems. *Discrete Event Dynamic Systems: Theory and Applications 7*, 2 (1997).

[16] HOLT, R. C. Comments on prevention of system deadlocks. *CACM 14*, 1 (1971).

[17] HOLT, R. C. Some deadlock properties of computer systems. *ACM Comput. Surv. 4*, 3 (1972).

[18] Intel C++ STM Compiler, Prototype Edition, Jan. 2008.

[19] IORDACHE, M. V., AND ANTSAKLIS, P. J. *Supervisory Control of Concurrent Systems: A Petri Net Structural Approach*. Birkhäuser, 2006.

[20] JULA, H., AND CANDEA, G. A scalable, sound, eventually-complete algorithm for deadlock immunity. In *Workshop on Runtime Verification* (2008).

[21] KAVI, K. M., MOSHTAGHI, A., AND YI CHEN, D. Modeling multithreaded applications using Petri nets. *International Journal of Parallel Programming 30*, 5 (2002).

[22] KNUTH, D. E. Additional comments on a problem in concurrent programming control. *CACM 9*, 5 (1966).

[23] LANDI, W. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst. 1*, 4 (1992).

[24] LANG, S.-D. An extended banker's algorithm for deadlock avoidance. *IEEE Trans. Software Eng 25*, 3 (1999).

[25] LARUS, J., AND RAJWAR, R. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[26] LEE, E. A. The problem with threads. Tech. rep., UC Berkeley EE & CS Department, Jan. 2006.

[27] LI, Z., ZHOU, M., AND WU, N. A survey and comparison of Petri net-based deadlock prevention policies for flexible manufacturing systems. *IEEE Trans. on Systems, Man, and Cybernetics—Part C 38*, 2 (2008).

[28] LU, S., PARK, S., SEO, E., AND ZHOU, Y. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS* (2008).

[29] MILLER, M. S. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

[30] MINOURA, T. Deadlock avoidance revisited. *J. ACM 29*, 4 (1982).

[31] MOODY, J. O., AND ANTSAKLIS, P. J. *Supervisory Control of Discrete Event Systems Using Petri Nets*. Kluwer Academic Publishers, 1998.

[32] MURATA, T. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE 77*, 4 (1989).

[33] NEWTON, G. Deadlock prevention, detection, and resolution: an annotated bibliography. *SIGOPS Oper. Syst. Rev. 13*, 2 (1979).

[34] NI, Y., WELC, A., ADL-TABATABAI, A.-R., BACH, M., BERKOWITS, S., COWNIE, J., GEVA, R., KOZHUKOW, S., NARAYANASWAMY, R., PREIS, J. O. S., SAHA, B., TAL, A., AND TIAN, X. Design and implementation of transactional constructs for C/C++. In *OOPSLA* (2008).

[35] NIR-BUCHBINDER, Y., TZOREF, R., AND UR, S. Deadlocks: from exhibiting to healing. In *Workshop on Runtime Verification* (2008).

[36] OpenIMPACT. `http://www.gelato.uiuc.edu/`.

[37] OpenLDAP Issue Tracking System. `http://www.openldap.org/its/`.

[38] QIN, F., TUCEK, J., ZHOU, Y., AND SUNDARESAN, J. Rx: Treating bugs as allergies—safe method to survive software failures. *ACM TOCS 25*, 3 (2007).

[39] REVELIOTIS, S. A. *Real-Time Management of Resource Allocation Systems: A Discrete-Event Systems Approach*. Springer, 2005.

[40] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS 15*, 4 (1997).

[41] SCHROEDER, B., WIERMAN, A., AND HARCHOL-BALTER, M. Open versus closed: A cautionary tale. In *NSDI* (2006).

[42] SPEAR, M. F., SILVERMAN, M., DALESSANDRO, L., MICHAEL, M. M., AND SCOTT, M. L. Implementing and exploiting inevitability in software transactional memory. In *Int'l. Conf. on Parallel Processing* (2008).

[43] SUN. *WorkShop: Command-Line Utilities*. Sun Press, 2006, ch. 24: Using Lock Lint.

[44] SUTTER, H., AND LARUS, J. Software and the concurrency revolution. *ACM Queue 3*, 7 (2005).

[45] TRICAS, F., COLOM, J. M., AND EZPELETA, J. Some improvements to the banker's algorithm based on the process structure. In *IEEE Int'l. Conf. on Robotics and Automation* (2000).

[46] WALLACE, C., JENSEN, P., AND SOPARKAR, N. Supervisory control of workflow scheduling. In *Proc. Int'l. Workshop on Advanced Transaction Models and Architectures* (1996).

[47] WANG, Y., KELLY, T., KUDLUR, M., MAHLKE, S., AND LAFORTUNE, S. The application of supervisory control to deadlock avoidance in concurrent software. In *Workshop on Discrete Event Systems* (2008).

[48] WANG, Y., KELLY, T., AND LAFORTUNE, S. Discrete control for safe execution of IT automation workflows. In *EuroSys* (2007).

[49] WELC, A., SAHA, B., AND ADL-TABATABAI, A.-R. Irrevocable transactions and their applications. In *SPAA* (2008).

[50] ZÖBEL, D., AND KOCH, C. Resolution techniques and complexity results with deadlocks: a classifying and annotated bibliography. *SIGOPS Oper. Syst. Rev. 22*, 1 (1988).

Deadlock Immunity: Enabling Systems To Defend Against Deadlocks

Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, George Candea
Dependable Systems Laboratory
EPFL
Switzerland

Abstract

Deadlock immunity is a property by which programs, once afflicted by a given deadlock, develop resistance against future occurrences of that and similar deadlocks. We describe a technique that enables programs to automatically gain such immunity without assistance from programmers or users. We implemented the technique for both Java and POSIX threads and evaluated it with several real systems, including MySQL, JBoss, SQLite, Apache ActiveMQ, Limewire, and Java JDK. The results demonstrate effectiveness against real deadlock bugs, while incurring modest performance overhead and scaling to 1024 threads. We therefore conclude that deadlock immunity offers programmers and users an attractive tool for coping with elusive deadlocks.

1 Introduction

Writing concurrent software is one of the most challenging endeavors faced by software engineers, because it requires careful reasoning about complex interactions between concurrently running threads. Many programmers consider concurrency bugs to be some of the most insidious and, not surprisingly, a large number of bugs are related to concurrency [16].

The simplest mechanism used for synchronizing concurrent accesses to shared data is the mutex lock. When threads do not coordinate correctly in their use of locks, deadlock can ensue—a situation whereby a group of threads cannot make forward progress, because each one is waiting to acquire a lock held by another thread in that group. Deadlock immunity helps develop resistance against such deadlocks.

Avoiding the introduction of deadlock bugs during development is challenging. Large software systems are developed by multiple teams totaling hundreds to thousands of programmers, which makes it hard to maintain the coding discipline needed to avoid deadlock bugs. Testing, although helpful, is not a panacea, because exercising all possible execution paths and thread interleavings is still infeasible in practice for all but toy programs.

Even deadlock-free code is not guaranteed to execute free of deadlocks once deployed in the field. Dependencies on deadlock-prone third party libraries or runtimes can deadlock programs that are otherwise correct. Upgrading these libraries or runtimes can introduce new executions that were not covered by prior testing. Furthermore, modern systems accommodate extensions written by third parties, which can introduce new deadlocks into the target systems (e.g., Web browser plugins, enterprise Java beans).

Debugging deadlocks is hard—merely seeing a deadlock happen does not mean the bug is easy to fix. Deadlocks often require complex sequences of low-probability events to manifest (e.g., timing or workload dependencies, presence or absence of debug code, compiler optimization options), making them hard to reproduce and diagnose. Sometimes deadlocks are too costly to fix, as they entail drastic redesign. Patches are error-prone: many concurrency bug fixes either introduce new bugs or, instead of fixing the underlying bug, merely decrease the probability of occurrence [16].

We expect the deadlock challenge to persist and likely become worse over time: On the one hand, software systems continue getting larger and more complex. On the other hand, owing to the advent of multi-core architectures and other forms of parallel hardware, new applications are written using more threads, while existing applications achieve higher degrees of runtime concurrency. There exist proposals for making concurrent programming easier, such as transactional memory [8], but issues surrounding I/O and long-running operations make it difficult to provide atomicity transparently.

In this paper, we introduce the notion of deadlock immunity—a property by which programs, once afflicted by a given deadlock, develop resistance against future occurrences of similar deadlocks. We describe Dimmunix, a tool for developing deadlock immunity with no assistance from programmers or users. The first time a deadlock pattern manifests, Dimmunix automatically captures its signature and subsequently avoids entering the same pattern. Signatures can be proactively distributed to immunize users who have not yet encountered that deadlock. Dimmunix can be used by customers to defend against deadlocks while waiting for a vendor patch, and by software vendors as a safety net.

In the rest of the paper we survey related work (§2), provide an overview of our system (§3-§4), give details of our technique (§5), describe three Dimmunix implementations (§6), evaluate them (§7), discuss how Dimmunix can be used in practice (§8), and conclude (§9).

2 Related Work

There is a spectrum of approaches for avoiding deadlocks, from purely static techniques to purely dynamic ones. Dimmunix targets general-purpose systems, not real-time or safety-critical ones, so we describe this spectrum of solutions keeping our target domain in mind.

Language-level approaches [3, 15] use powerful type systems to simplify the writing of lock-based concurrent programs and thus avoid synchronization problems altogether. This avoids runtime performance overhead and prevents deadlocks outright, but requires programmers to be disciplined, adopt new languages and constructs, or annotate their code. While this is the ideal way to avoid deadlocks, programmers' human limits have motivated a number of complementary approaches.

Transactional memory (TM) [8] holds promise for simplifying the way program concurrency is expressed. TM converts the locking order problem into a thread scheduling problem, thus moving the burden from programmers to the runtime, which we consider a good tradeoff. There are still challenges with TM semantics, such as what happens when programmers use large atomic blocks, or when TM code calls into non-TM code or performs I/O. Performance is still an issue, and [14] shows that many modern TM implementations use lock-based techniques to improve performance and are subject to deadlock. Thus, we believe TM is powerful, but it cannot address all concurrency problems in real systems.

Time-triggered systems [13] and statically scheduled real-time systems [22] perform task synchronization before the program runs, by deciding schedules a priori based on task parameters like mutual-exclusion constraints and request processing time. When such parameters are known a priori, the approach guarantees safety and liveness; however, general-purpose systems rarely have such information ahead of time. Event-triggered real-time systems are more flexible and incorporate a priori constraints in the form of thread priorities; protocols like priority ceiling [20], used to prevent priority inversion, conveniently prevent deadlocks too. In general-purpose systems, though, even merely assigning priorities to the various threads is difficult, as the threads often serve a variety of purposes over their lifetime.

Static analysis tools look for deadlocks at compile time and help programmers remove them. ESC [7] uses a theorem prover and relies on annotations to provide knowledge to the analysis; Houdini [6] helps generate some of these annotations automatically. [5] and [21] use flow-sensitive analyses to find deadlocks. In Java JDK 1.4, the tool described in [21] reported 100,000 potential deadlocks and the authors used unsound filtering to trim this result set down to 70, which were then manually reduced to 7 actual deadlock bugs. Static analyses run fast, avoid runtime overheads, and can help prevent deadlocks, but when they generate false positives, it is ultimately the programmers who have to winnow the results. Developers under pressure to ship production code fast are often reticent to take on this burden.

Another approach to finding deadlocks is to use model checkers, which systematically explore all possible states of the program; in the case of concurrent programs, this includes all thread interleavings. Model checkers achieve high coverage and are sound, but suffer from poor scalability due to the "state-space explosion" problem. Java PathFinder, one of the most successful model checkers, is restricted to applications up to $\sim$10 KLOC [10] and does not support native I/O libraries. Real-world applications are large (e.g., MySQL has >1 MLOC) and perform frequent I/O, which restricts the use of model checking in the development of general-purpose systems.

Further toward the dynamic end of the spectrum, [17] discovers deadlocks at runtime, then wraps the corresponding parts of the code in one "gate lock"; in subsequent executions, the gate lock must be acquired prior to entering the code block. This approach is similar to [2], except that the latter detects deadlocks statically, thus exhibiting more false positives than [17]. In a dual approach to these two, [23] modifies the JVM to serialize threads' access to lock sets (instead of program code) that could induce deadlocks. Dimmunix shares ideas with these dynamic approaches, but uses added context information to achieve finer grain avoidance and considerably fewer false positives (as will be seen in §7.3).

Finally, there are purely dynamic approaches, like Rx [18]. Upon deadlock, Rx can roll back a program to a checkpoint and retry the execution in a modified environment; new timing conditions could prevent deadlock reoccurrence. However, Rx does not (and was not meant to) build up resistance against future occurrences of the deadlock, so the system as a whole does not "improve" itself over time. The performance overhead induced by repeated re-executions can be unpredictable (in the extreme case of a deterministic deadlock, Rx cannot go past it) and retried executions cannot safely span I/O. In contrast, Dimmunix actively prevents programs from re-encountering previously seen deadlock patterns.

Deadlock immunity explores a new design point on this spectrum of deadlock avoidance solutions, combining static elements (e.g., control flow signatures) with dynamic approaches (e.g., runtime steering of thread schedules). This combination makes Dimmunix embody new tradeoffs, which we found to be advantageous when avoiding deadlocks in large, real, general-purpose systems.

3 System Overview

Programs augmented with a deadlock immunity system develop "antibodies" matching observed deadlock patterns, store them in a persistent history, and then alter future thread schedules in order to avoid executing patterns like the ones that were previously seen. With every new deadlock pattern encountered by the program, its resistance to deadlocks is improved.

When buggy code runs and deadlocks, we refer to an approximate suffix of the call flow that led to deadlock as a *deadlock pattern*—this is an approximation of the control flow that turned the bug into a deadlock. A runtime instantiation of a deadlock pattern constitutes a *deadlock occurrence*. Thus, a deadlock bug begets a deadlock pattern, which in turn begets a deadlock occurrence. One deadlock pattern can generate a potentially unbounded number of runtime deadlock occurrences, e.g., because lock identities vary across different manifestations of the same deadlock pattern. Dimmunix automatically avoids previously seen deadlock patterns, in order to reduce the number of deadlock occurrences. To recognize repeated deadlock patterns, it saves "fingerprints" of every new pattern; we call these *deadlock signatures*. Runtime conditions can cause a deadlock pattern to not always lead to deadlock, in which case avoiding the pattern results in a false positive (more details in §5.5).

The Dimmunix architecture is illustrated in Figure 1. There are two parts: *avoidance instrumentation code* prevents reoccurrences of previously encountered deadlocks and a *monitor thread* finds and adds deadlock information to the persistent *deadlock history*. Avoidance code can be directly instrumented into the target binary or can reside in a thread library. This instrumentation code intercepts the lock/unlock operations in target programs and transfers control to Dimmunix any time lock or unlock is performed; Dimmunix itself runs within the address space of the target program.

At the beginning of a lock call, a *request* method in the avoidance instrumentation decides whether to allow the lock operation to proceed. This decision can be *GO*, if locking is allowed, or *YIELD*, if not. In the case of a yield, the thread is forced by the instrumentation code to yield the CPU, and the lock attempt is transparently retried later. When the program finally acquires the lock, the instrumentation code invokes *acquired*. Unlock operations are preceded by a call to *release*.

The avoidance code enqueues request, go, yield, acquired, and release events onto a lock-free queue that is drained by the monitor thread. The monitor wakes up periodically and updates a resource allocation graph (RAG) according to received events, searches for deadlock cycles, and saves the cycle signatures to the persistent history. The delay between the occurrence of a deadlock and its detection by the asynchronous monitor has an upper bound determined by the wakeup frequency.

Dimmunix uses the RAG to represent a program's synchronization state. Most edges are labeled with the call stack of the origin thread, representing an approximation of that thread's recent control flow. When a deadlock is found, Dimmunix archives a combination of the involved threads' stacks into a deadlock signature.

Avoiding deadlocks requires anticipating whether the acquisition of a lock would lead to the instantiation of a signature of a previously-encountered deadlock pattern. For a signature with call stacks $\{S_1, S_2,...\}$ to be instantiated, there must exist threads $T_1, T_2,...$ that either hold or are allowed to wait for locks $L_1, L_2,...$ while having call stacks $S_1, S_2,...$ An instantiation of a signature captures the corresponding thread-lock-stack bindings: $\{(T_1, L_1, S_1), (T_2, L_2, S_2), ...\}$.

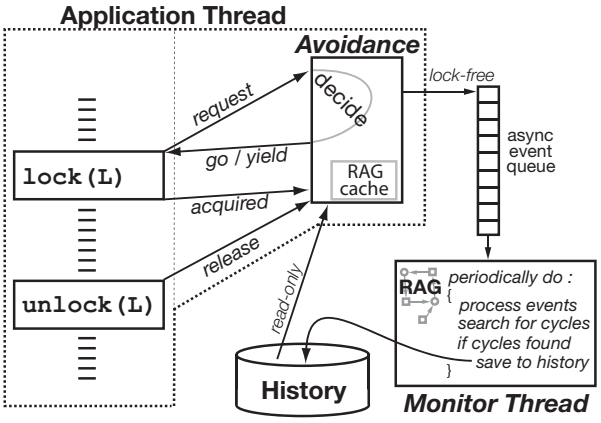


Figure 1: Dimmunix architecture.

The way in which a deadlocked program *recovers* is orthogonal to Dimmunix and, in practice, would most likely be done via restart. Dimmunix can provide a hook in the monitor thread for programs to define more sophisticated deadlock recovery methods; the hook can be invoked right after the deadlock signature is saved. For instance, plugging Rx's checkpoint/rollback facility [18] into this application-specific deadlock resolution hook could provide application-transparent deadlock recovery.

Any scheduling-based approach to deadlock avoidance faces the risk of occasionally reaching starvation states, in which threads are actively yielding, waiting in vain for synchronization conditions to change. In Dimmunix, this is handled automatically: when induced starvation occurs, Dimmunix saves the signature of the starvation state, breaks the starvation by canceling the yield for the starved thread holding most locks, and allows the freed thread to pursue its most recently requested lock. Dimmunix will subsequently be able to avoid entering this same starvation condition again.

We recommend Dimmunix for general-purpose systems, such as desktop and enterprise applications, server software, etc.; in real-time systems or safety-critical systems, Dimmunix can cause undue interference (§5.7). Systems in which even the very first occurrence of a deadlock cannot be tolerated are not good targets for Dimmunix; such systems require more programmer-intensive approaches if they want to run deadlock-free.

Dimmunix can be used by software vendors and end users alike. Faced with the current impossibility of shipping large software that is bug-free, vendors could instrument their ready-to-ship software with Dimmunix and get an extra safety net. Dimmunix will keep users happy by allowing them to use the deadlock-prone system while developers try to fix the bugs. Also, users frustrated with

deadlock-prone applications can use Dimmunix on their own to improve their user experience. We do not advocate deadlock immunity as a replacement for correct concurrent programming—ultimately, concurrency bugs need to be fixed in the design and the code—but it does offer a "band-aid" with many practical benefits.

4 An Example

We now illustrate how Dimmunix works with an example of two deadlock-prone threads.

The pseudocode on the right accesses two global shared variables A and B, each protected by its own mutex. s_1, s_2,... are the equivalent of goto labels. For simplicity, we assume there are no pointers.

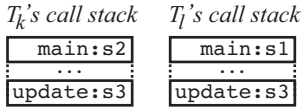
```
     main {
s1:    update(A,B)
       ...
s2:    update(B,A)
     }

     update(x,y) {
s3:    lock(x)
s4:    lock(y)
       ...
       unlock(y)
       unlock(x)
     }
```

If two different threads T_i and T_j run the code concurrently, they may attempt to lock A and B in opposite order, which can lead to deadlock, i.e., if T_i executes statement s_1 and then s_3, while T_j executes s_2 followed by s_3. The call flow can be represented abstractly as $<T_i{:}[s_1, s_3], T_j{:}[s_2, s_3]>$. There exist other execution patterns too, such as $<T_i{:}[s_1, s_3], T_j{:}[s_1, s_3]>$ that do not lead to deadlock.

The first time the code enters a deadlock, Dimmunix will see it as a cycle in the RAG and save its signature based on the threads' call stacks at the time of their lock acquisitions. When T_i acquires the lock on A, the return addresses on its stack are $[s_1, s_3]$, because it called *update()* from s_1 and *lock()* from s_3; similarly, when T_j acquires the lock on B, T_j's call stack is $[s_2, s_3]$. Upon deadlock, the $<T_i{:}[s_1, s_3], T_j{:}[s_2, s_3]>$ call pattern is saved to history as deadlock signature $\{[s_1, s_3], [s_2, s_3]\}$. Signatures do not include thread or lock identities, thus making them portable from one execution to the next.

Consider now a subsequent run of the program, in which some thread T_k executes s_2 followed by s_3, acquires the lock on B, and then some other thread T_l executes s_1 and then makes the call to *lock(x)* in statement s_3, in order to acquire A. If Dimmunix were to allow this lock operation to proceed, this execution could deadlock. Dimmunix infers the deadlock danger by matching the threads' call stacks (shown on right) to the signature.



Given that there is a match, Dimmunix decides to force T_l to yield until the lock that T_k acquired at s_3 is released. After B is released, T_l is allowed to lock A and proceed. In this way, the program became immune to the deadlock pattern $\{[s_1, s_3], [s_2, s_3]\}$.

Note that Dimmunix does not merely serialize code blocks, as would be done by wrapping *update()* in a Java *synchronized*{...} block or as was done in prior work. For instance, on the above example, [17] would add a "gate lock" around the code for *update()* and serialize all calls to it, even in the case of execution patterns that do not lead to deadlock, such as $\{[s_1, s_3], [s_1, s_3]\}$. [23] would add a "ghost lock" for A and B, that would have to be acquired prior to locking either A or B.

Dimmunix achieves finer grain avoidance by (a) using call path information to distinguish between executions—of all paths that end up at s_3, Dimmunix avoids only those that executed a call path previously seen to lead to deadlock—and (b) using runtime information about which locks are held by other threads to avoid these paths only when they indeed seem dangerous.

5 Deadlock and Starvation Avoidance

We now present selected details of Dimmunix: the core data structure (§5.1), detection (§5.2), construction of signatures (§5.3), runtime avoidance of archived signatures (§5.4), calibration of signature matching precision (§5.5), auxiliary data structures (§5.6), and a synopsis of Dimmunix's properties and limitations (§5.7).

5.1 Capturing Synchronization State

Dimmunix conceptually uses a resource allocation graph (RAG) to represent the synchronization state of a program. In practice, the RAG is built on top of several performance-optimized data structures (details in §5.6).

The RAG is a directed graph with two types of vertices: *threads* T and *locks* L. There are three types of edges connecting threads to locks and one type of edges connecting threads to threads. *Request edges* indicate that a thread T wants to acquire lock L, *allow edges* indicate that thread T has been allowed by Dimmunix to block waiting for L, and *hold edges* indicate that T has acquired and presently holds lock L. If the avoidance code decides to not allow a thread T's lock request, it will force T to yield. This state is reflected in the RAG by a *yield edge* connecting thread T to T', indicating that T is currently yielding because of locks that T' acquired or was allowed to wait for. Dimmunix reschedules the paused thread T whenever lock conditions change in a way that could enable T to acquire the desired lock. Figure 2 illustrates a subgraph of a real RAG.

A hold edge, like $L_7 \xrightarrow{S_y} T_{13}$, always points from a lock to a thread and indicates that the lock is held by that thread; it also carries as label a simplified version S_y of the call stack that the thread had at the time it acquired the lock. A yield edge, like $T_{13} \xrightarrow{S_x} T_{22}$, always points from a thread to another thread; it indicates that T_{13} has been forced to yield because T_{22} acquired a lock with call stack S_x that would cause T_{13} to instantiate a signature if it was allowed to execute lock().

In order to support reentrant locks, as are standard in Java and available in POSIX threads, the RAG is a multi-

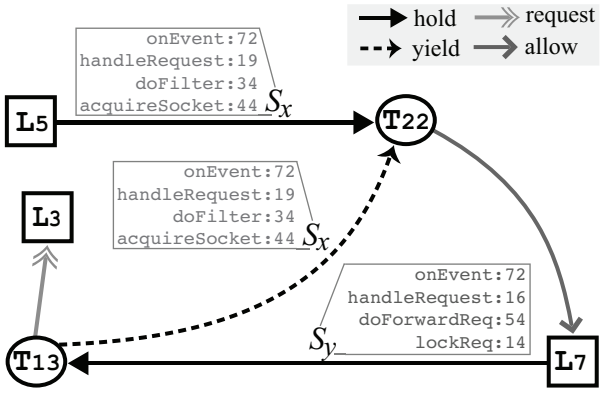


Figure 2: Fragment of a resource allocation graph.

set of edges; it can represent locks that are acquired multiple times by the same holder and, thus, have to be released as many times as acquired before becoming available to other threads.

Finally, the RAG does not always provide an up-to-date view of the program's synchronization state, since it is updated lazily by the monitor. This is acceptable for cycle detection, but the avoidance code needs some information to always be current, such as the mapping from locks to owners. Therefore, the avoidance instrumentation also maintains a simpler "cache" of parts of the RAG (in the form of simplified lock-free data structures) to make correct yield/allow decisions.

5.2 Detecting Deadlocks and Starvation

The monitor thread wakes up every τ milliseconds, drains all events from the lock-free event queue, and updates the RAG according to these events; then it searches for cycles. The monitor only searches for cycles involving edges that were added by the most recently processed batch of events; there cannot be new cycles formed that involve exclusively old edges. The value of τ is configurable, and the right choice depends on the application at hand; e.g., in an interactive program, $\tau = 100$ milliseconds would be reasonable.

Events enqueued by the same thread are correctly ordered with respect to each other. As far as other threads are concerned, we need to ensure a partial ordering that guarantees a *release* event on lock L in thread T_i will appear in the queue prior to any other thread T_j's *acquired* event on L. Given that the runtime (e.g., JVM) completes lock(L) in T_j strictly after it completed unlock(L) in T_i, and the *release* event in T_i precedes the unlock(L), and the *acquired* event in T_j follows the lock(L), the required partial ordering is guaranteed.

There are two cycle types of interest: deadlock cycles and yield cycles. A thread T is in a *deadlock* iff T is part of a cycle made up exclusively of hold, allow, and request edges—this is similar to deadlock cycles in standard wait-for graphs. Yield cycles are used to detect induced starvation. Any yield-based deadlock avoidance technique runs the risk of inducing one or more threads to starve due to yielding while waiting for a thread that is blocked. Thus, any dynamic scheduling-based deadlock avoidance approach must also avoid induced starvation.

Consider Figure 3, which shows a RAG in which a starvation state has been reached (nodes and edges not used in the discussion are smaller and dotted; call stack edge labels are not shown). For T_1 to be starved, both its yield edges $T_1 \rightarrow T_2$ and $T_1 \rightarrow T_3$ must be part of cycles, as well as both of T_4's yield edges. If the RAG had only the $(T_1, T_2, ..., T_1)$ and $(T_1, T_3, L, T_4, T_6, ..., T_1)$ cycles, then this would not be a starvation state, because T_4 could evade starvation through T_5, allowing T_1 to eventually evade through T_3. If, as in Figure 3, cycle $(T_1, T_3, L, T_4, T_5, ..., T_1)$ is also present, then neither thread can make progress.

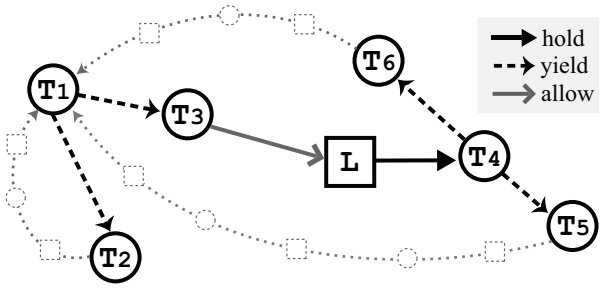


Figure 3: Starved threads in a yield cycle.

We say that a thread T is involved in an *induced starvation* condition iff T is part of a yield cycle. A yield cycle is a subgraph of the RAG in which all nodes reachable from a node T through T's yield edges can in turn reach T. The graph in Figure 3 is a yield cycle.

Dimmunix uses cycle detection as a universal mechanism for detecting both deadlocks and induced starvation: when the monitor encounters a yield cycle in the RAG, it saves its signature to the history, as if it was a deadlock. Dimmunix uses the same logic to avoid both deadlock patterns and induced starvation patterns.

5.3 From Cycles to Signatures

The signature of a cycle is a multiset containing the call stack labels of all hold edges and yield edges in that cycle. The signature must be a multiset because different threads may have acquired different locks while having the same call stack, by virtue of executing the same code.

Figure 2 shows a simple yield cycle $(T_{13}, T_{22}, L_7, T_{13})$, whose signature is $\{S_x, S_y\}$. The signature is archived by the monitor into the history that persists across program restarts.

A signature contains one call stack per thread blocked in the detected deadlock or starvation. The number of

threads involved is bounded by the maximum number of threads that can run at any given time, so a signature can have no more than that number of call stacks. A call stack is always of finite size (usually set by the OS or thread library); thus, the size of a signature is finite. Signatures are essentially permutations of "instruction addresses" in the code, and there is a finite number of instructions in an application; given that duplicate signatures are disallowed, the signature history cannot grow indefinitely.

After saving a deadlock signature, the monitor can wait for deadlock recovery to be performed externally, or it can invoke an application-specific deadlock resolution handler. After saving a starvation signature, the monitor can break the starvation as described in §3.

Signatures abstract solely the call flow that led to deadlock or starvation; no program data (such as lock/thread IDs or values of variables) are recorded. This ensures that signatures preserve the generality of a deadlock pattern and are fully portable from one execution to the next. Program state can vary frequently within one run or from one run to the next (due to inputs, load, etc.) and requiring that this state also be matched, in addition to the call flow, would cause Dimmunix to incur many false negatives. The downside of the generality of patterns are occasional false positives, as will be discussed in §5.5.

5.4 Avoiding Previously Seen Patterns

The avoidance code is split from the monitoring code, so that expensive operations (cycle detection, history file I/O, etc.) can be done asynchronously, outside the application's critical path. The deadlock history is loaded from disk into memory at startup time and shared read-only among all threads; the monitor is the only thread mutating the history, both in-memory and on-disk.

As illustrated in Figure 1, Dimmunix intercepts all lock and unlock calls in the target program or intercepts them within a thread library. When the application performs a lock, the instrumentation invokes the *request* method, which returns a *YIELD* or *GO* answer. The *GO* case indicates that Dimmunix considers it safe (w.r.t. the history) for the thread to block waiting for the lock; this does not mean the lock is available. When the lock is actually acquired, the instrumentation invokes an *acquired* method in the avoidance code; when the lock is released, it invokes a *release* method—both methods serve solely to update the RAG, as they do not return any decision.

The *request* method determines whether allowing the current thread T's lock request would take the program into a situation that matches a previously seen deadlock or starvation. For this, it tentatively adds the corresponding allow edge to the RAG cache and searches for an instantiation of a signature from history; this consists of finding a set of (thread, lock, stack) tuples in the RAG cache that provide an exact cover of the signature. All thread-lock-stack tuples in the instance must correspond to distinct threads and locks. Checking for signature instantiation takes into consideration allow edges in addition to hold edges, because an allow edge represents a commitment by a thread to block waiting for a lock.

If a potential deadlock instance is found, then the tentative allow edge is flipped around into a request edge, and a yield edge is inserted into the RAG cache from T to each thread $T_i \neq T$ in the signature instance: these threads T_i are the "causes" of T's yield. Each yield edge gets its label from its yield cause (e.g., in Figure 2, $T_{13} \rightarrow T_{22}$ gets label S_x from hold edge $L_5 \stackrel{S_x}{\rightarrow} T_{22}$). A *yield* event is sent to the monitor and a *YIELD* decision is returned to the instrumentation.

If no instance is found, then T's allow edge is kept, the corresponding *allow* event is sent to the monitor, and a *GO* decision is returned; any yield edges emerging from the current thread's node are removed.

When the *acquired* method is invoked, the corresponding allow edge in the RAG cache is converted into a hold edge and an *acquired* event is sent to the monitor. When *release* is invoked, the corresponding hold edge is removed and a *release* event is enqueued for the monitor.

Dimmunix provides two levels of immunity, each with its pros and cons; they can be selected via a configuration flag. By default, *weak immunity* is enforced: induced starvation is automatically broken (after saving its signature) and the program continues as if Dimmunix wasn't present—this is the least intrusive, but may lead to reoccurrences of some deadlock patterns. The number of times the initially avoided deadlock pattern can reoccur is bounded by the maximum nesting depth of locks in the program. The intuitive reason behind this upper bound is that avoiding a deadlock or starvation is always done at least one nesting level above the one where the avoided deadlock or starvation occurs. In *strong immunity* mode, the program is restarted every time a starvation is encountered, instead of merely breaking the yield cycle—while more intrusive, this mode guarantees that no deadlock or starvation patterns ever reoccur.

In our experience, one deadlock bug usually has one deadlock pattern (see §7). In the ideal case, if there are n deadlock bugs in the program, after exactly n occurrences of deadlocks the program will have acquired immunity against all n bugs. However, there could also be k induced starvation cases and, in the worst case, each new starvation situation will lead (after breaking) to the deadlock that was being avoided. Thus, it will take $n+k$ occurrences of deadlocks to develop immunity against all n deadlocks in the program. The exact values of n and k depend on the specific program at hand.

5.5 Calibrating the Matching Precision

A signature contains the call stacks from the corresponding RAG cycle, along with a "matching depth," indicating how long a suffix of each call stack should be considered during matching. In the simplest case, this depth is

set to a fixed value (4 by default). However, choosing too long a suffix can cause Dimmunix to miss manifestations of a deadlock bug, while choosing too short a suffix can lead to mispredicting a runtime call flow as being headed for deadlock (i.e., this is a false positive). In this section we describe how Dimmunix can optionally calibrate the matching depth at runtime.

First, Dimmunix must be able to heuristically determine whether it did not cause a false positive (FP), i.e., whether forcing a thread to yield indeed avoided a deadlock or not. After deciding to avoid a given signature X, Dimmunix performs a retrospective analysis: All lock operations performed by threads involved in the potential deadlock are logged to the monitor thread, along with lock operations performed by the blocked thread after it was released from the yield. The monitor thread then looks for lock inversions in this log; if none are found, the avoidance was likely a FP, i.e., in the absence of avoidance, there would have likely not been a deadlock.

Using this heuristic, Dimmunix estimates the FP rate for each possible matching depth: when signature X is created, depth starts at 1 and is kept there for the first N_A avoidances of X, then incremented for the next N_A avoidances of X, and so on until maximum depth is reached. The N_A parameter is 20 by default. Then Dimmunix chooses the smallest depth d that exhibited the lowest FP rate $FP_{\min}$ and sets X's matching depth to d.

False positives are not exclusively due to overly general matching, but could also be caused by input or value dependencies; e.g., pattern X may lead to deadlock for some inputs but not for others, so avoiding X can have false positives even at the most precise matching depth. For this reason, $FP_{\min}$ can be non-zero, and multiple depths can have the same $FP_{\min}$ rate; choosing the smallest depth gives us the most general pattern.

The algorithm implemented in Dimmunix is slightly more complex. For instance, to increase calibration speed, when encountering a FP at depth k, Dimmunix analyzes whether it would have performed avoidance had the depth been $k+1$, $k+2$,... and, if yes, increments the FP counts for those depths as well; this allows the calibration to run fewer than N_A iterations at the larger depths. One could associate a per-stack matching depth instead of a per-signature depth; while this would be theoretically more precise, we found the current heuristic to be satisfactory for the systems discussed in §7.

Once X's calibration is complete, Dimmunix stops tracking FPs for X. After X has been avoided N_T times, Dimmunix performs a recalibration, in case program conditions have changed ($N_T = 10^4$ by default).

Dynamic calibration is a way to heuristically choose a deadlock pattern that is more balanced than if we chose a fixed-length suffix of the call stacks. This optional calibration algorithm is orthogonal to the rest of Dimmunix, since avoiding an execution pattern that matches partially a signature will cause all executions that match the signature fully (i.e., the precise deadlock pattern) to be avoided. Calibration merely makes Dimmunix more efficient at avoiding deadlocks similar to the ones already encountered, without incurring undue false positives.

5.6 Auxiliary Data Structures

The RAG is extended with several other data structures, which serve to improve both asymptotic and actual performance. For example, we achieve $O(1)$ lookup of thread and lock nodes, because they are kept in a preallocated vector and a lightly loaded hash table, respectively. In the case of library-based Dimmunix, the RAG nodes are embedded in the library's own thread and mutex data structures. Moreover, data structures necessary for avoidance and detection are themselves embedded in the thread and lock nodes. For example, the set *yieldCause* containing all of a thread T's yield edges is directly accessible from the thread node T.

Dimmunix uses a hash table to map raw call stacks to our own call stack objects. Matching a call stack consists of hashing the raw call stack and finding the corresponding metadata object S, if it exists. From S, one can directly get to, e.g., the *Allowed* set, containing handles to all the threads that are permitted to wait for locks while having call stack S; *Allowed* includes those threads that have acquired and still hold the locks. When checking for signature instantiations, a thread will check the *Allowed* sets for all call stacks S_i from the signature to be matched. In most cases, at least one of these sets is empty, meaning there is no thread holding a lock in that stack configuration, so the signature is not instantiated.

Complexity of the *request* method in the avoidance code is $O(D \cdot H \cdot T! \cdot G^T)$, where D is the maximum depth at which Dimmunix can match a call stack, H is the number of signatures in history, T is the maximum number of threads that can be involved in a deadlock, and G is the maximum number of locks acquired or waited for at the same time by threads with the exact same call stack. In practice, D is a constant and T is almost always two [16], bringing complexity closer to $O(H \cdot G^2)$.

Most accesses to the history and RAG cache are thread-safe, because they mutate allow and hold edges that involve the current thread only, so no other thread could be changing them at the same time. The *request* and *release* methods are the only ones that need to both consult and update the shared *Allowed* set. To do so safely without using locks, we use a variation of Peterson's algorithm for mutual exclusion generalized to n threads [9].

To find cycles in the RAG, we use the Colored DFS algorithm, whose theoretical complexity is $O(N_E + N_T \cdot (|V| + |E|))$, where the RAG is a graph $[V, E]$, the maximum number of threads issuing lock requests at any one time is N_T, and the maximum number of events in the asynchronous lock-free event queue is N_E.

5.7 Dimmunix Properties and Limitations

In this section, we summarize the most important properties of the algorithms presented so far. A formal description of an earlier version of the algorithm and its properties can be found in [12].

Dimmunix can never affect a deadlock-free program's correctness. Dimmunix saves a signature only when a deadlock actually happens, i.e., when there is a cycle in the RAG. A program that never deadlocks will have a perpetually empty history, which means no avoidance will ever be done.

Dimmunix must know about all synchronization mechanisms used in the system. In programs that mix Dimmunix-instrumented synchronization with non-instrumented synchronization, Dimmunix can interfere with the mechanisms it is unaware of (e.g., a program that combines non-instrumented busy-wait loops with instrumented POSIX threads locks could be starved). Thus, Dimmunix requires that the non-instrumented synchronization routines be indicated in a configuration file, similar to the way RacerX [5] does; Dimmunix will then ignore the avoidance decision whenever a call to the foreign synchronization is encountered.

Some deadlock patterns are too risky to avoid. Say there is an operation W that is implemented such that all possible execution paths are deadlock-prone. Dimmunix essentially prunes those paths that have deadlocked in the past, leaving only those that have not deadlocked; for W, this could mean eventually pruning all execution paths, leading to the loss of W's functionality. Although we have never noticed such functionality loss in thousands of executions of several instrumented desktop and server programs, it is possible in principle, so Dimmunix offers two options when running in "weak immunity" mode:

First, Dimmunix allows users to disable signatures. Every time Dimmunix avoids a signature, it logs the avoidance action in a field of the signature in the history. Now consider the following use scenario: a user is in front of their Web browser and, every time a suspected deadlock is avoided, Dimmunix beeps, the way pop-up blockers do. Say the user clicks on a menu item and s/he just hears a beep but nothing happens—the menu has been disabled due to avoidance. The user can now instruct Dimmunix to disable the last avoided signature, the same way s/he would enable pop-ups for a given site. The signature will never be avoided again and the menu is usable again (but it may occasionally deadlock, since the deadlock pattern is not being avoided).

Second, if users cannot be directly involved in detecting such starvation-based loss of functionality, Dimmunix has a configurable system-wide upper bound (e.g., 200 msec) for how long Dimmunix can keep a thread waiting in order to avoid a deadlock pattern; once this maximum is reached, the thread is released from the yield. Once a particular pattern has accumulated a large number of such aborts, it can be automatically disabled, or a warning can be issued instead to the user indicating this deadlock pattern is too risky to avoid.

Dimmunix cannot induce a non-real-time program to produce wrong outputs, even with strong immunity, because Dimmunix works solely by altering thread schedules. Schedulers in general-purpose systems (like a commodity JVM) do not provide strong guarantees, so the correctness of a program's outputs cannot reasonably depend on the scheduler. Starvation, as described above, is a liveness issue in a non-real-time system, so it cannot lead to the generation of incorrect outputs, i.e., it cannot violate safety.

Dimmunix never adds a false deadlock to the history, since it detects and saves only signatures of true deadlock patterns. Without a real deadlock, there cannot be a deadlock cycle in the RAG, hence the signature database cannot contain the signature of a deadlock pattern that never led to deadlock.

6 Dimmunix for Java and POSIX Threads

We currently have three implementations of Dimmunix: one for Java programs and two for programs using POSIX threads (pthreads): one for FreeBSD libthr and the other for Linux NPTL. They can be downloaded from `http://dimmunix.epfl.ch/`. The Java version has $\sim$1400 lines of Java code. The FreeBSD version has $\sim$1100 lines of C++ code plus $\sim$20 lines changed in libthr, while the Linux version has $\sim$1700 lines of C++ code plus $\sim$30 lines changed in NPTL. The latter's extra code is to support both 32-bit and 64-bit platforms.

The implementations illustrate two different approaches: the Java version directly instruments the target bytecode, while the pthreads implementations rely on modified pthreads libraries. Neither approach requires access to a program's source code nor does it ask programmers to changes how they write their code.

Java provides three main synchronization primitives: monitors, explicit locks, and wait queues; our implementation currently supports the first two. Monitors are declared using a *synchronized(x)* {...} statement, which translates, at the bytecode level, into *monitorenter(x)*, followed by the code block, followed by *monitorexit(x)*; an additional *monitorexit(x)* is placed on the exception path. If a thread attempts to enter a monitor it is already in, the call returns immediately; the thread will have to exit that monitor the same number of times it entered it before the monitor becomes available to others.

In order to intercept the monitor entry/exit and explicit lock/unlock requests, we use an aspect-oriented compiler, AspectJ [1], to directly instrument target programs at either bytecode or source level. The instrumented Java bytecode can be executed in any standard Java 1.5 VM or later. We implemented the avoidance code as aspects that get woven into the target program before and after every *monitorenter* and *lock* bytecode, as well as before every *monitorexit* and *unlock* bytecode. The aspects intercept

the corresponding operations, update all necessary data structures, and decide whether to allow a lock request or to pause the thread instead. Call stacks are vectors of <methodName, file:line#> strings. The monitor thread is started automatically when the program starts up.

Dimmunix pauses a thread by making a Java *wait* call from within the instrumentation code; we do not use *Thread.yield*, because we found *wait* to scale considerably better. There is one synchronization object, *yieldLock[T]*, dedicated to each thread T and, when T is to yield, the instrumentation code calls *yieldLock[t].wait()*. When a thread T' frees a lock L' acquired with call stack S', then all threads T_i for which $(T', L', S') \in$ *yieldCause[*T_i*]* (see §5.6) have no more reason to yield, so they are woken up via a call to *yieldLock[*T_i*].notifyAll()*.

For **POSIX threads**, we modified the `libthr` library in FreeBSD and the Native POSIX Threads Library (NPTL) in Linux, which is part of glibc. The modified libraries are 100% compatible drop-in replacements. Porting to other POSIX threads libraries is straightforward. The pthreads-based implementations are similar to the Java implementation, with a few exceptions:

The basic synchronization primitive in POSIX threads is the *mutex*, and there are three types: normal mutex, recursive mutex (equivalent to Java's reentrant lock), and error-checking mutex, which returns EDEADLK if a thread attempts to lock a non-recursive locked mutex and thus self-deadlock. Dimmunix does not watch for self-deadlocks, since pthreads already offers the error-checking mutex option.

We instrumented all the basic mutex management functions. Locks associated with conditional variables are also instrumented. Having direct access to the thread library internals simplifies data access; for example, instead of keeping track of locks externally (as is done in the Java version), we can simply extend the original library data structures. Call stacks are unwound with backtrace(), and Dimmunix computes the byte offset of each return address relative to the beginning of the binary and stores these offsets in execution-independent signatures.

Java guarantees that all native lock() operations are blocking, i.e., after a successful *request* the thread will either *acquire* the lock or become blocked on it. This is not the case for pthreads, which allows a lock acquisition to time out (*pthread_mutex_timedlock*) or to return immediately if there is contention (*pthread_mutex_trylock*). To support trylocks and timedlocks, we introduced a new event in pthreads Dimmunix called *cancel*, which rolls back a previous lock *request* upon a timeout.

7 Evaluation

In this section we answer a number of practical questions. First and foremost, does Dimmunix work for real systems that do I/O, use system libraries, and interact with users and other systems (§7.1)? What performance overhead does Dimmunix introduce, and how does this overhead vary as parameters change (§7.2)? What is the impact of false positives on performance (§7.3)? What overheads does Dimmunix introduce in terms of resource consumption (§7.4)?

We evaluated Dimmunix with several real systems: MySQL (C/C++ open-source database), SQLite (C open-source embedded database), Apache ActiveMQ (Java open-source message broker for enterprise applications), JBoss (Java open-source enterprise application server), Limewire (Java peer-to-peer file sharing application), the Java JDK (provider of all class libraries that implement the Java API), and HawkNL (C library specialized for network games). These are widely-used systems within their category; some are large, such as MySQL, which has over 1 million lines of code excluding comments.

For all experiments reported here, we used strong immunity, with τ=100 msec; in the microbenchmarks we used a fixed call stack matching depth of 4. Measurements were obtained on 8-core computers (2x4-core Intel Xeon E5310 1.6GHz CPUs), 4GB RAM, WD-1500 hard disk, two NetXtreme II GbE interfaces with dedicated GbE switch, running Linux and FreeBSD, Java HotSpot Server VM 1.6.0, and Java SE 1.6.0.

7.1 Effectiveness Against Real Deadlocks

In practice, deadlocks arise from two main sources: bugs in the logic of the program (§7.1.1) and technically permissible, but yet inappropriate uses of third party code (§7.1.2); Dimmunix addresses both.

7.1.1 True Deadlock Bugs

To verify effectiveness against real bugs, we reproduced deadlocks that were reported against the systems described above. We used timing loops to generate "exploits," i.e. test cases that deterministically reproduced the deadlocks. It took on average two programmer-days to successfully reproduce a bug; we abandoned many bugs, because we could not reproduce them reliably. We ran each test 100 times in three different configurations: First, we ran the unmodified program, and the test always deadlocked prior to completion. Second, we ran the program instrumented with full Dimmunix, but ignored all yield decisions, to verify that timing changes introduced by the instrumentation did not affect the deadlock—again, each test case deadlocked in every run. Finally, we ran the program with full Dimmunix, with signatures of previously-encountered deadlocks in the history—in each case, Dimmunix successfully avoided the deadlock and allowed the test to run to completion.

The results are centralized in Table 1. We include the number of yields recorded during the trials with full Dimmunix as a measure of how often deadlock patterns were encountered and avoided. For most cases, there is one yield, corresponding to the one deadlock reproduced

System	Bug #	Deadlock Between ...	# Yields per Trial			Dlk Patterns	
			Min	Avg	Max	#	Depth
MySQL 6.0.4	37080	INSERT and TRUNCATE in two different threads	1	1	4	1	4
SQLite 3.3.0	1672	Deadlock in the custom recursive lock implementation	1	1	1	1	3
HawkNL 1.6b3	n/a	nlShutdown() called concurrently with nlClose()	10	10	10	1	2
MySQL 5.0 JDBC	2147	PreparedStatement.getWarnings() and Connection.close()	1	1	1	1	3
MySQL 5.0 JDBC	14972	Connection.prepareStatement() and Statement.close()	1	1	1	1	4
MySQL 5.0 JDBC	31136	PreparedStatement.executeQuery() and Connection.close()	1	1	1	1	3
MySQL 5.0 JDBC	17709	Statement.executeQuery() and Connection.prepareStatement()	1	1	1	1	3
Limewire 4.17.9	1449	HsqlDB TaskQueue cancel and shutdown()	15	15	15	2	10,10
ActiveMQ 3.1	336	Listener creation and active dispatching of messages to consumer	1	181079	221292	1	2
ActiveMQ 4.0	575	Queue.dropEvent() and PrefetchSubscription.add()	11252	80387	113652	3	2,2,2

Table 1: A few reported deadlock bugs avoided by Dimmunix in popular server and desktop applications.

by the test case. In some cases, however, the number of yields was much higher, because avoiding the initial deadlock enabled the test to continue and re-enter the same deadlock pattern later. For all but the ActiveMQ tests there were no false positives; in the case of ActiveMQ, we could not accurately determine if any of the reported yields were false positives.

We also inspected the code for each bug, to determine how many different deadlock patterns can be generated by the bug. The last two columns in Table 1 indicate the number of deadlock patterns ("#" column) and the size of the pattern ("Depth" column). Depth corresponds to the type of matching depth discussed in §5.5. Dimmunix correctly detected, saved, and avoided all patterns, except in the case of ActiveMQ #575, where we were able to only reproduce one of the three patterns, so Dimmunix only witnessed, saved and avoided that one.

7.1.2 Invitations to Deadlock

When using third party libraries, it is possible to use the offered APIs in ways that lead to deadlock inside the library, despite there being no logic bug in the calling program. For example, several synchronized base classes in the Java runtime environment can lead to deadlocks.

Consider two vectors v_1, v_2 in a multithreaded program—since *Vector* is a synchronized class, programmers allegedly need not be concerned by concurrent access to vectors. However, if one thread wants to add all elements of v_2 to v_1 via $v_1.addAll(v_2)$, while another thread concurrently does the reverse via $v_2.addAll(v_1)$, the program can deadlock inside the JDK, because under the covers the JDK locks v_1 then v_2 in one thread, and v_2 then v_1 in the other thread. This is a general problem for all synchronized *Collection* classes in the JDK.

Table 2 shows deadlocks we reproduced in JDK 1.6.0; they were all successfully avoided by Dimmunix. While not bugs per se, these are invitations to deadlock. Ideally, APIs should be documented thoroughly, but there is always a tradeoff between productivity and pedantry in documentation. Moreover, programmers cannot think of every possible way in which their API will be used. Runtime tools like Dimmunix provide an inexpensive alternative to this dilemma: avoid the deadlocks when and if they manifest. This requires no programmer intervention and no JDK modifications.

PrintWriter class: With w a PrintWriter, concurrently call w.write() and CharArrayWriter.writeTo(w)
Vector: Concurrently call v_1.addAll(v_2) and v_2.addAll(v_1)
Hashtable: With h_1 a member of h_2 and h_2 a member of h_1, concurrently call h_1.equals(foo) and h_2.equals(bar)
StringBuffer: With StringBuffers s_1 and s_2, concurrently call s_1.append(s_2) and s_2.append(s_1)
BeanContextSupport: concurrent propertyChange() and remove()

Table 2: Java JDK 1.6 deadlocks avoided by Dimmunix.

7.2 Performance Overhead

In this section we systematically quantify Dimmunix's impact on system performance, using request throughput and latency as the main metrics. First, we report in §7.2.1 end-to-end measurements on real systems and then use synthetic microbenchmarks to drill deeper into the performance characteristics (§7.2.2).

7.2.1 Real Applications

To measure end-to-end overhead, we ran standard performance benchmarks on "immunized" JBoss 4.0 and MySQL 5.0 JDBC. For JBoss, we used the RUBiS e-commerce benchmark [19], for MySQL JDBC we used JDBCBench [11]. For HawkNL, Limewire, and ActiveMQ we are unaware of any benchmarks.

We took the measurements for various history sizes, to see how overhead changes as more signatures accumulate. Since we had insufficient real deadlock signatures, we synthesized additional ones as random combinations of real program stacks with which the target system performs synchronization. From the point of view of avoidance overhead, synthesized signatures have the same effect as real ones. Figure 4 presents the results.

The cost of immunity against up to 128 deadlock signatures is modest in large systems with hundreds of threads in realistic settings—e.g., JBoss/RUBiS ran with

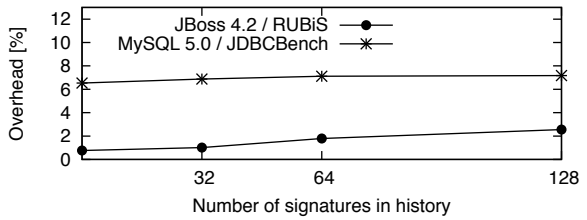


Figure 4: Performance overhead introduced in real systems, computed on the benchmark-specific metric. Maximum overhead is 2.6% for JBoss and 7.17% for MySQL JDBC.

3000 clients, a mixed read/write workload, and performed on average ∼500 lock operations per second while running 280 threads. We did not witness a statistically meaningful drop in response time for either system. In light of these results, it is reasonable to suggest that users encountering deadlocks be offered the option of using Dimmunix right away to cope, while the respective development teams fix the underlying bugs. The development teams themselves could also provide deadlock signatures to customers until fixes for the bugs become available.

7.2.2 Microbenchmarks

To dissect Dimmunix's performance behavior and understand how it varies with various parameters, we wrote a synchronization-intensive microbenchmark that creates N_t threads and has them synchronize on locks from a total of N_l locks shared among the threads; a lock is held for δ_{in} time before being released and a new lock is requested after δ_{out} time; the delays are implemented as busy loops, thus simulating computation done inside and outside the critical sections. The threads call multiple functions within the microbenchmark so as to build up different call stacks; which function is called at each level is chosen randomly, thus generating a uniformly distributed selection of call stacks.

We also wrote a tool that generates synthetic deadlock history files containing H signatures, all of size S; for a real application, H represents the number of deadlock/starvation signatures that have accumulated in the history, and a signature's size indicates the number of threads involved in that deadlock. Generated signatures consist of random stack combinations for synchronization operations in the benchmark program—not signatures of real deadlocks, but avoided as if they were.

Overhead as a function of the number of threads: Figure 5 shows how synchronization throughput (in terms of lock operations) varies with the number of threads in Java and pthreads, respectively. We chose δ_{in}=1 microsecond and δ_{out}=1 millisecond, to simulate a program that grabs a lock, updates some in-memory shared data structures, releases the lock, and then performs computation outside the critical section.

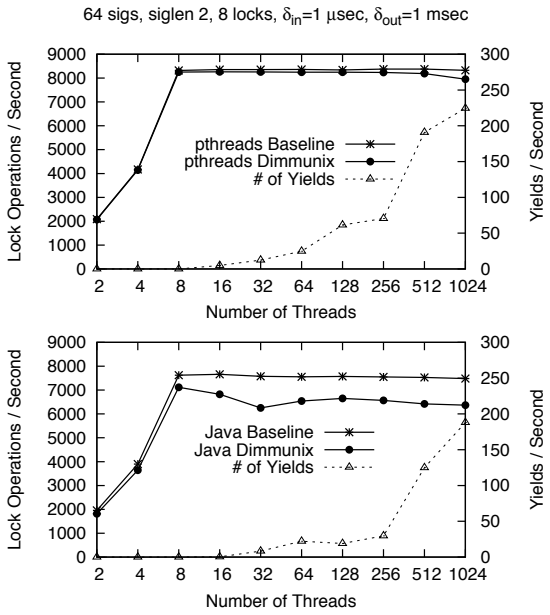


Figure 5: Dimmunix microbenchmark lock throughput as a function of number of threads. Overhead is 0.6% to 4.5% for FreeBSD pthreads and 6.5% to 17.5% for Java.

We observe that Dimmunix scales well: for up to 1024 threads, the pthreads implementation exhibits maximum 4.5% overhead, while the Java implementation maximum 17.5%. The difference between the implementations is, we believe, primarily due to Java-specific overheads (such as returning the call stack as a vector of strings vs. mere pointers in C, or introducing extra memory fences around synchronized{} blocks, that pthreads does not do). As the benchmark approaches the behavior we see in real applications that perform I/O, we would expect the overhead to be further absorbed by the time spent between lock/unlock operations. To validate this hypothesis, we measured the variation of lock throughput with the values of δ_{in} and δ_{out}—Figure 6 shows the results for Java; pthreads results are similar.

The overhead introduced by Dimmunix is highest when the program does nothing but lock and unlock (i.e., δ_{in}=δ_{out}=0). This is not surprising, because Dimmunix intercepts the calls to lock/unlock and performs additional computation in the critical path. lock/unlock are generally fast operations that take a few machine instructions to perform, so adding 10× more instructions in the path will cause the overhead to be 10×. However, as the interval between critical sections (δ_{out}) or inside critical sections (δ_{in}) increases, the throughput difference between the immunized vs. non-immunized microbenchmark decreases correspondingly. For most common scenarios (i.e., inter-critical-section intervals of 1 millisecond or more), overhead is modest.

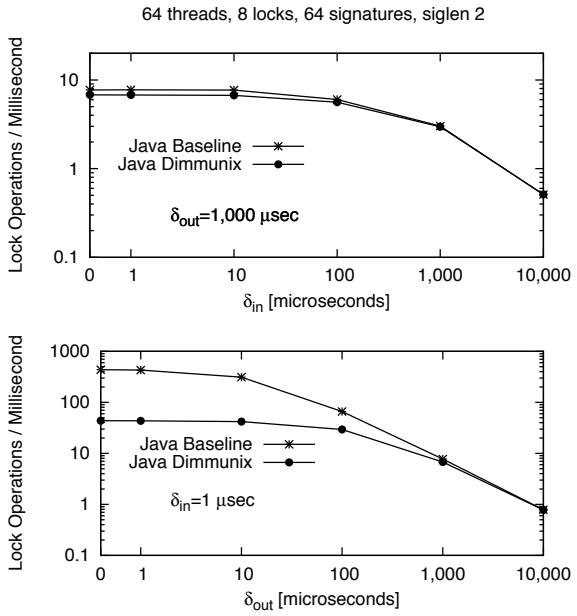


Figure 6: Variation of lock throughput as a function of δ_{in} and δ_{out} for Java; the pthreads version is similar.

Note that a direct comparison of overhead between Dimmunix and the baseline is somewhat unfair to Dimmunix, because non-immunized programs deadlock and stop running, whereas immunized ones continue running and doing useful work.

Impact of history size and matching depth: The performance penalty incurred by matching current executions against signatures from history should increase with the size of the history (i.e., number of signatures) as well as the depth at which signatures are matched with current stacks. Average length of a signature (i.e., average number of threads involved in the captured deadlock) also influences matching time, but the vast majority of deadlocks in practice are limited to two threads [16], so variation with signature size is not that interesting.

In addition to the matching overhead, as more and more deadlocks are discovered in the program, the program must be serialized increasingly more in order to be deadlock-safe (i.e., there are more deadlocks to avoid)—our overhead measurements include both effects.

We show in Figure 7 the performance overhead introduced by varying history size from 2-256 signatures. The overhead introduced by history size and matching depth is relatively constant across this range, which means that searching through history is a negligible component of Dimmunix overhead.

Breakdown of overhead: Having seen the impact of number of threads, history size, and matching depth, we profiled the overhead, to understand which parts of Dimmunix contribute the most. For this, we selectively disabled parts of Dimmunix and measured the lock

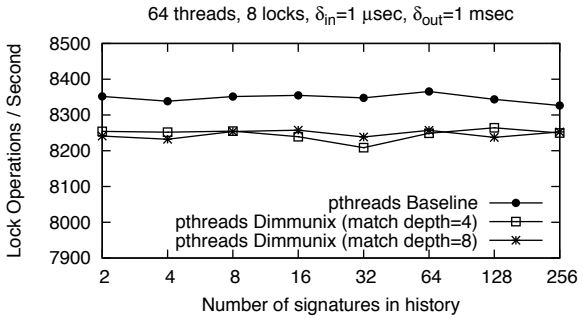


Figure 7: Lock throughput as a function of history size and matching depth for pthreads. Java results are similar.

throughput. First we measured the overhead introduced by the base instrumentation, then we added the data structure lookups and updates performed by *request* in the avoidance code, then we ran full Dimmunix, including avoidance.

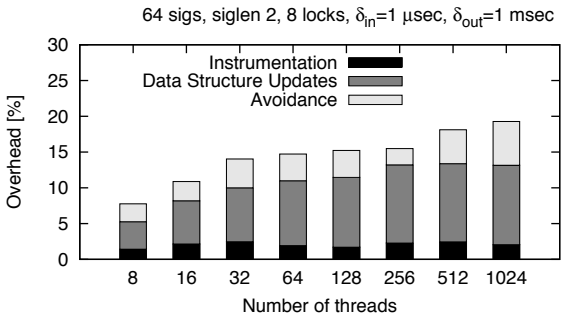


Figure 8: Breakdown of overhead for Java Dimmunix.

The results for Java are shown in Figure 8—the bulk of the overhead is introduced by the data structure lookups and updates. For pthreads, the trend is similar, except that the dominant fraction of overhead is introduced by the instrumentation code. The main reason is that the changes to the pthreads library interfere with the fastpath of the pthreads mutex: it first performs a compare-and-swap (CAS) and only if that is unsuccessful does it make a system call. Our current implementation causes that CAS to be unsuccessful with higher probability.

7.3 False Positives

Any approach that tries to predict the future with the purpose of avoiding bad outcomes suffers from false positives, i.e., wrongly predicting that the bad outcome will occur. Dimmunix is no exception. False positives can arise when signatures are matched too shallowly, or when the lock order in a pattern depends on inputs, program state, etc. Our microbenchmark does not have the latter type of dependencies.

In a false positive, Dimmunix reschedules threads in order to avoid an apparent impending deadlock that would actually not have occurred; this can have negative or positive effects on performance, the latter due to reduced contention. We concern ourselves here with the negative effects, which result from a loss in parallelism: Dimmunix serializes "needlessly" a portion of the program execution, which causes the program to run slower.

In our microbenchmark, let D be the program's maximum stack depth (we set D=10) and let k be the depth at which we match signature stacks in the avoidance code. We consider a true positive to be an avoidance triggered by a match at depth D; a false positive occurs when a signature is matched to depth k but would not match to depth D. If k=D, there are no false positives, because the signatures are matched exactly, but if k<D, then we can get false positives, because several different runtime stacks produce a match on the same signature.

In order to determine the overhead induced by false positives, we compare the lock throughput obtained while matching at depths k<D (i.e., in the presence of false positives) to that obtained while matching at depth D (no false positives)—the difference represents the time wasted due to false positives. To measure the overhead introduced by Dimmunix itself, separate from that introduced by false positives, we measure the overhead of Dimmunix when all its avoidance decisions are ignored (thus, no false positives) and subtract it from the baseline. Calibration of matching precision is turned off. Figure 9 shows the results—as the precision of matching is increased, the overhead induced by false positives decreases. There are hardly any false positives for depths of 8 and 9 because the probability of encountering a stack that matches at that depth and not at depth 10 is very low.

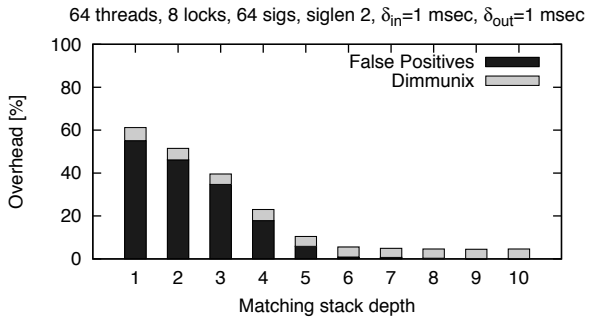


Figure 9: Overhead induced by false positives.

We ran this same experiment using the technique based on gate locks [17], the best hybrid dynamic/static deadlock avoidance we know of. To avoid the 64 deadlocks represented in history, 45 gate locks were required; since [17] does not use call stacks, matching depth is irelevant. Throughput overhead with gate locks was 70%—more than an order of magnitude greater than Dimmunix's 4.6% overhead for stack depth ≥8 and close to Dimmunix's 61.2% overhead at stack depth 1. There were 561,627 false positives with gate locks; in contrast, Dimmunix's false positives ranged from 0 (at depth 10) to 573,912 (at depth 1). This is consistent with the fact that, for stack depth 1, the two approaches are similar.

As mentioned in §5.7, false positives can also disable functionality. We did not encounter such loss during any of the thousands of executions of the various server and desktop applications described in this paper, but Dimmunix does provide two resolutions for these situations: manual or automatic disabling of signatures.

7.4 Resource Utilization

The final aspect of Dimmunix we wish to measure is how many additional resources it requires, compared to non-immunized programs. Dimmunix uses CPU time for computation, memory for its data structures, and disk space for the history. The latter is insignificant: on the order of 200-1000 bytes per signature, amounting to tens of KBs for a realistic history. CPU overhead was virtually zero in all our measurements; in fact, delaying some of the threads can even lead to negative overhead, through the reduction of contention.

In measurements ranging from 2-1024 threads, 8-32 shared locks, and a history of 64 two-thread signatures, the pthreads implementations incurred a memory overhead of 6-25 MB, and the Java implementation 79-127 MB. As described in §5.6, we use preallocation to reduce performance overhead, and the data structures themselves have redundancy, to speed up lookups. We expect a Dimmunix version optimized for memory footprint to have considerably less memory overhead.

8 Using Dimmunix in Practice

Dimmunix helps programs develop resistance against deadlocks, without assistance from programmers (i.e., no annotations, no specifications) or from system users. Dimmunix can be used as a band-aid to complement all the other development and deployment tools for software systems, such as static analyzers and model checkers. In systems that have checkpoint facilities [18] or are microrebootable [4], strong immunity can offer strong guarantees at low cost; in other general-purpose systems, weak immunity lends progressively stronger resistance to deadlocks, without incurring additional recoveries.

Aside from achieving immunity, Dimmunix can also be used as an alternative to patching and upgrading: instead of modifying the program code, it can be "patched" against deadlock bugs by simply inserting the corresponding bug's signature into the deadlock history and asking Dimmunix to reload the history; the target program need not even be restarted. Similarly, vendors could ship their software with signatures for known deadlocks, as an alternative to fixing them in the released code when doing so is too expensive or risky.

When upgrading a system, the signature history also needs to be updated. First, code locations captured in the signatures' call stacks may have shifted or disappeared; static analysis can be used to map from old to new code and "port" signatures from one revision to the next, thus accounting for refactoring, addition/removal of code, etc. Second, at a higher semantic level, deadlock behaviors may have been modified by the upgrade (e.g., by fixing a bug), thus rendering some signatures obsolete, regardless of porting. The calibration of matching precision (§5.5) is therefore re-enabled after every upgrade for all signatures, and any signatures that encounter 100% false positive rate after this recalibration can be automatically discarded as obsolete.

Dimmunix currently does not take into account the priorities of threads when making scheduling decisions; the server applications we are familiar with do not use thread priorities, but perhaps other applications do. We believe support for priorities can easily be added.

Although Dimmunix does not introduce new bugs, avoiding deadlocks could trigger latent program bugs that might otherwise not manifest, in much the same way an upgrade of the JVM, system libraries, or kernel could do. While this is not a limitation per se, it is a factor to be considered when using Dimmunix in practice.

9 Conclusion

In this paper we described a technique for augmenting software systems with an "immune system" against deadlocks. Unlike pure deadlock avoidance, deadlock immunity is easier to achieve in practical settings and appears to be almost as useful as ideal pure avoidance. The essence of our approach is to "fingerprint" control flows that lead to deadlock, save them in a persistent history, and avoid them during subsequent runs.

We showed empirically that real systems instrumented with Dimmunix can develop resistance against real deadlocks, without any assistance from programmers or users, while incurring modest overhead. Dimmunix scales gracefully to large software systems with up to 1024 threads and preserves correctness of general-purpose programs. We conclude that deadlock immunity is a practical approach to avoiding deadlocks, that can improve end users' experience with deadlock-prone systems and also keep production systems running deadlock-free despite the bugs that lurk within.

Acknowledgments

We thank our shepherd, Wolfgang Schröder-Preikschat, and the anonymous reviewers for their help in refining our paper. We are indebted to Vinu Rajashekar for the NPTL port, Silviu Andrica for his help with ActiveMQ, and Katerina Argyraki, Emmanuel Cecchet, Steve Hand, and Willy Zwaenepoel for their early feedback and ideas.

References

[1] Aspectj. http://www.eclipse.org/aspectj.

[2] P. Boronat and V. Cholvi. A transformation to provide deadlock-free programs. In *Intl. Conf. on Computational Science*, 2003.

[3] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *17th Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

[4] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – a technique for cheap recovery. In *Proc. 6th Symp. on Operating Systems Design and Implementation*, 2004.

[5] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *19th ACM Symp. on Operating Systems Principles*, 2003.

[6] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proc. Intl. Symp. on Formal Methods Europe*, 2001.

[7] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Conf. on Programming Language Design and Implementation*, 2002.

[8] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *20th Intl. Symp. on Computer Architecture*, 1993.

[9] M. Hofri. Proof of a mutual exclusion algorithm. *ACM SIGOPS Operating Systems Review*, 24(1):18–22, 1990.

[10] Java PathFinder. http://javapathfinder.sourceforge.net, 2007.

[11] JDBCBench. http://mmmysql.sourceforge.net/performance/.

[12] H. Jula and G. Candea. A scalable, sound, eventually-complete algorithm for deadlock immunity. In *Proc. 8th Workshop on Runtime Verification*, 2008.

[13] H. Kopetz. Fault management in the time triggered protocol. In *Proc. EuroMicro*, 1994.

[14] E. Koskinen and M. Herlihy. Dreadlocks: Efficient deadlock detection. In *Proc. 20th ACM Symp. on Parallelism in Algorithms and Architectures*, 2008.

[15] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2), 1980.

[16] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study on real world concurrency bug characteristics. In *Proc. 13th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[17] Y. Nir-Buchbinder, R. Tzoref, and S. Ur. Deadlocks: from exhibiting to healing. In *Proc. 8th Workshop on Runtime Verification*, 2008.

[18] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies – a safe method to survive software failures. *ACM Trans. on Computer Systems*, 25(3), 2007.

[19] RUBiS. http://rubis.objectweb.org, 2007.

[20] L. Sha, R. Rajkumar, and S. Sathaye. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. on Computers*, 39(9):1175–1185, 1990.

[21] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *19th European Conf. on Object-Oriented Programming*, 2005.

[22] J. Xu and D. Parnas. Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Trans. on Software Engineering*, 16(3), 1990.

[23] F. Zeng and R. P. Martin. Ghost locks: Deadlock prevention for Java. In *Mid-Atlantic Student Workshop on Programming Languages and Systems*, 2004.

Difference Engine: Harnessing Memory Redundancy in Virtual Machines

Diwaker Gupta, Sangmin Lee*, Michael Vrable,
Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat
{dgupta,mvrable,savage,snoeren,varghese,voelker,vahdat}@cs.ucsd.edu
University of California, San Diego

Abstract

Virtual machine monitors (VMMs) are a popular platform for Internet hosting centers and cloud-based compute services. By multiplexing hardware resources among virtual machines (VMs) running commodity operating systems, VMMs decrease both the capital outlay and management overhead of hosting centers. Appropriate placement and migration policies can take advantage of statistical multiplexing to effectively utilize available processors. However, main memory is not amenable to such multiplexing and is often the primary bottleneck in achieving higher degrees of consolidation.

Previous efforts have shown that content-based page sharing provides modest decreases in the memory footprint of VMs running similar operating systems and applications. Our studies show that significant additional gains can be had by leveraging both sub-page level sharing (through page patching) and in-core memory compression. We build *Difference Engine*, an extension to the Xen virtual machine monitor, to support each of these—in addition to standard copy-on-write full page sharing—and demonstrate substantial savings not only between VMs running similar applications and operating systems (up to 90%), but even across VMs running disparate workloads (up to 65%). In head-to-head memory-savings comparisons, Difference Engine outperforms VMware ESX server by a factor of 1.5 for homogeneous workloads and by a factor of 1.6–2.5 for heterogeneous workloads. In all cases, the performance overhead of Difference Engine is less than 7%.

1 Introduction

Virtualization technology has improved dramatically over the past decade to become pervasive within the service-delivery industry. Virtual machines are particularly attractive for server consolidation. Their strong resource and fault isolation guarantees allow multiplexing of hardware among individual services, each with (potentially) distinct software configurations. Anecdotally, individual server machines often run at 5–10% CPU utilization. Operators' reasons are manifold: because of the need to over-provision for peak levels of demand, because fault isolation mandates that individual services run on individual machines, and because many services often run best on a particular operating system configuration. The promise of virtual machine technology for server consolidation is to run multiple services on a single physical machine while still allowing independent configuration and failure isolation.

While physical CPUs are frequently amenable to multiplexing, main memory is not. Many services run comfortably on a machine with 1 GB of RAM; multiplexing ten VMs on that same host, however, would allocate each just 100 MB of RAM. Increasing a machine's physical memory is often both difficult and expensive. Incremental upgrades in memory capacity are subject to both the availability of extra slots on the motherboard and the ability to support higher-capacity modules: such upgrades often involve replacing—as opposed to just adding—memory chips. Moreover, not only is high-density memory expensive, it also consumes significant power. Furthermore, as many-core processors become the norm, the bottleneck for VM multiplexing will increasingly be the memory, not the CPU. Finally, both applications and operating systems are becoming more and more resource intensive over time. As a result, commodity operating systems require significant physical memory to avoid frequent paging.

Not surprisingly, researchers and commercial VM software vendors have developed techniques to decrease the memory requirements for virtual machines. Notably, the VMware ESX server implements content-based page sharing, which has been shown to reduce the memory footprint of multiple, homogeneous virtual machines by 10–40% [24]. We find that these values depend greatly on the operating system and configuration of the guest VMs. We are not aware of any previously published sharing figures for mixed-OS ESX deployments. Our evaluation indicates, however, that the benefits of ESX-style page sharing decrease as the heterogeneity of the guest VMs increases, due in large part to the fact that page sharing requires the candidate pages to be *identical*.

The premise of this work is that there are significant additional benefits from sharing at a sub-page granularity, i.e., there are many pages that are *nearly* identical. We show that it is possible to efficiently find such similar pages and to coalesce them into a much smaller memory footprint. Among the set of similar pages, we are able to

*Currently at UT Austin, sangmin@cs.utexas.edu

store most as *patches* relative to a single baseline page.

Finally, we also compress those pages that are unlikely to be accessed in the near future. Traditional stream-based compression algorithms typically do not have sufficient "look-ahead" to find commonality across a large number of pages or across large chunks of content, but they can exploit commonality within a local region, such as a single memory page. We show that an efficient implementation of compression nicely complements page sharing and patching.

In this paper, we present Difference Engine, an extension to the Xen VMM [6] that not only shares identical pages, but also supports sub-page sharing and in-memory compression of infrequently accessed pages. Our results show that Difference Engine can reduce the memory footprint of homogeneous workloads by up to 90%, a significant improvement over previously published systems [24]. For a heterogeneous setup (different operating systems hosting different applications), we can reduce memory usage by nearly 65%. In head-to-head comparisons against VMware's ESX server running the same workloads, Difference Engine delivers a factor of 1.5 more memory savings for a homogeneous workload and a factor of 1.6-2.5 more memory savings for heterogeneous workloads.

Critically, we demonstrate that these benefits can be obtained without negatively impacting application performance: in our experiments across a variety of workloads, Difference Engine imposes less than 7% overhead. We further show that Difference Engine can leverage improved memory efficiency to increase aggregate system performance by utilizing the free memory to create additional virtual machines in support of a target workload. For instance, one can improve the aggregate throughput available from multiplexing virtual machines running Web services onto a single physical machine.

2 Related Work

Difference Engine builds upon substantial previous work in page sharing, delta encoding and memory compression. In each instance, we attempt to leverage existing approaches where appropriate.

2.1 Page Sharing

Two common approaches in the literature for finding redundant pages are content-based page sharing, exemplified by VMWare's ESX server [24], and explicitly tracking page changes to build knowledge of identical pages, exemplified by "transparent page sharing" in Disco [9]. Transparent page sharing can be more efficient, but requires several modifications to the guest OS, in contrast to ESX server and Difference Engine which require no modifications.

To find sharing candidates, ESX hashes contents of each page and uses hash collisions to identify potential duplicates. To guard against false collisions, both ESX server and Difference Engine perform a byte-by-byte comparison before actually sharing the page.

Once shared, our system can manage page updates in a copy-on-write fashion, as in Disco and ESX server. We build upon earlier work on *flash cloning* [23] of VMs, which allows new VMs to be cloned from an existing VM in milliseconds; as the newly created VM writes to its memory, it is given private copies of the shared pages. An extension by Kloster *et al.* studied page sharing in Xen [13] and we build upon this experience, adding support for fully virtualized (HVM) guests, integrating the global clock, and improving the overall reliability and performance.

2.2 Delta Encoding

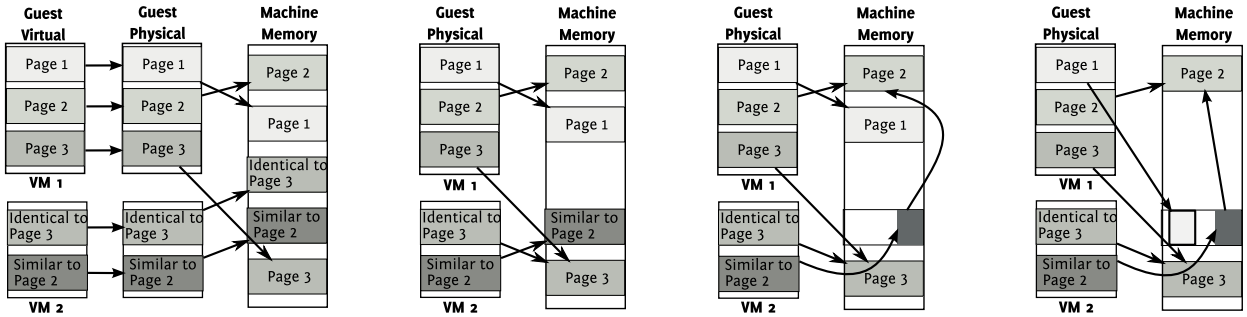
Our initial investigations into page similarity were inspired by research in leveraging similarity across files in large file systems. In GLIMPSE [18], Manber proposed computing Rabin fingerprints over fixed-size blocks at multiple offsets in a file. Similar files will then share some fingerprints. Thus the maximum number of common fingerprints is a strong indicator of similarity. However, in a dynamically evolving virtual memory system, this approach does not scale well since every time a page changes its fingerprints must be recomputed as well. Further, it is inefficient to find the maximal intersecting set from among a large number of candidate pages.

Broder adapted Manber's approach to the problem of identifying documents (in this case, Web pages) that are nearly identical using a combination of Rabin fingerprints and sampling based on minimum values under a set of random permutations [8]. His paper also contains a general discussion of how thresholds should be set for inferring document similarity based on the number of common fingerprints or sets of fingerprints.

While these techniques can be used to identify similar files, they do not address how to efficiently encode the differences. Douglis and Iyengar explored using Rabin fingerprints and delta encoding to compress similar files in the DERD system [12], but only considered whole files. Kulkarni *et al.* [14] extended the DERD scheme to exploit similarity at the block level. Difference Engine also tries to exploit memory redundancy at several different granularities.

2.3 Memory Compression

In-memory compression is not a new idea. Douglis *et al.* [11] implemented memory compression in the Sprite operating system with mixed results. In their experience, memory compression was sometimes beneficial, but at other times the performance overhead outweighed the memory savings. Subsequently, Wilson *et al.* argued



(a) Initial (b) Page sharing (c) Patching (d) Compression

Figure 1: The three different memory conservation techniques employed by Difference Engine: page sharing, page patching, and compression. In this example, five physical pages are stored in less than three machine memory pages for a savings of roughly 50%.

Douglis' mixed results were primarily due to slow hardware [25]. They also developed new compression algorithms (leveraged by Difference Engine) that exploited the inherent structure present in virtual memory, whereas earlier systems used general-purpose compression algorithms.

Despite its mixed history, several operating systems have dabbled with in-memory compression. In the early 90s, a Macintosh application, Ram Doubler, promised to "double a machine's RAM" [15]. Tuduce *et al.* [22] implemented a compressed cache for Linux that adaptively manages the amount of physical memory devoted to compressed pages using a simple algorithm shown to be effective across a wide variety of workloads.

3 Architecture

Difference Engine uses three distinct mechanisms that work together to realize the benefits of memory sharing, as shown in Figure 1. In this example, two VMs have allocated five pages total, each initially backed by distinct pages in machine memory (Figure 1(a)). For brevity, we only show how the mapping from guest physical memory to machine memory changes; the guest virtual to guest physical mapping remains unaffected. First, for identical pages across the VMs, we store a single copy and create references that point to the original. In Figure 1(b), one page in VM-2 is identical to one in VM-1. For pages that are similar, but not identical, we store a patch against a reference page and discard the redundant copy. In Figure 1(c), the second page of VM-2 is stored as a patch to the second page of VM-1. Finally, for pages that are unique and infrequently accessed, we compress them in memory to save space. In Figure 1(d), the remaining private page in VM-1 is compressed. The actual machine memory footprint is now less than three pages, down from five pages originally.

In all three cases, efficiency concerns require us to select candidate pages that are unlikely to be accessed in the near future. We employ a global clock that scans memory in the background, identifying pages that have not been recently used. In addition, reference pages for sharing or patching must be found quickly without introducing performance overhead. Difference Engine uses full-page hashes and hash-based fingerprints to identify good candidates. Finally, we implement a demand paging mechanism that supplements main memory by writing VM pages to disk to support overcommitment, allowing the total memory required for all VMs to temporarily exceed the physical memory capacity.

3.1 Page Sharing

Difference Engine's implementation of content-based page sharing is similar to those in earlier systems. We walk through memory looking for identical pages. As we scan memory, we hash each page and index it based on its hash value. Identical pages hash to the same value and a collision indicates that a potential matching page has been found. We perform a byte-by-byte comparison to ensure that the pages are indeed identical before sharing them.

Upon identifying target pages for sharing, we reclaim one of the pages and update the virtual memory to point at the shared copy. Both mappings are marked read-only, so that writes to a shared page cause a page fault that will be trapped by the VMM. The VMM returns a private copy of the shared page to the faulting VM and updates the virtual memory mappings appropriately. If no VM refers to a shared page, the VMM reclaims it and returns it to the free memory pool.

3.2 Patching

Traditionally, the goal of page sharing has been to eliminate redundant copies of *identical* pages. Difference Engine considers further reducing the memory required to store *similar* pages by constructing patches that represent a page as the difference relative to a reference page. To motivate this design decision, we provide an initial study into the potential savings due to sub-page sharing, both within and across virtual machines. First, we define the

following two heterogeneous workloads, each involving three 512-MB virtual machines:

- MIXED-1: Windows XP SP1 hosting RUBiS [10]; Debian 3.1 compiling the Linux kernel; Slackware 10.2 compiling Vim 7.0 followed by a run of the `lmbench` benchmark [19].
- MIXED-2: Windows XP SP1 running Apache 2.2.8 hosting approximately 32,000 static Web pages crawled from Wikipedia, with `httperf` running on a separate machine requesting these pages; Debian 3.1 running the SysBench database benchmark [1] using 10 threads to issue 100,000 requests; Slackware 10.2 running dbench [2] with 10 clients for six minutes followed by a run of the IOZone benchmark [3].

We designed these workloads to stress the memory-saving mechanisms since opportunities for identical page sharing are reduced. Our choice of applications was guided by the VMmark benchmark [17] and the vmbench suite [20]. In this first experiment, for a variety of configurations, we suspend the VMs after completing a benchmark, and consider a static snapshot of their memory to determine the number of pages required to store the images using various techniques. Table 1 shows the results of our analysis for the MIXED-1 workload.

The first column breaks down these 393,120 pages into three categories: 149,038 zero pages (i.e., the page contains all zeros), 52,436 sharable pages (the page is not all zeros, and there exists at least one other identical page), and 191,646 unique pages (no other page in memory is exactly the same). The second column shows the number of pages required to store these three categories of pages using traditional page sharing. Each unique page must be preserved; however, we only need to store one copy of a set of identical pages. Hence, the 52,436 non-unique pages contain only 3577 distinct pages—implying there are roughly fourteen copies of every non-unique page. Furthermore, only one copy of the zero page is needed. In total, the 393,120 original pages can be represented by 195,224 distinct pages—a 50% savings.

The third column depicts the additional savings available if we consider sub-page sharing. Using a cut-off of 2 KB for the patch size (i.e., we do not create a patch if it will take up more than half a page), we identify 144,497 distinct pages eligible for patching. We store the 50,727 remaining pages as is and use them as reference pages for the patched pages. For each of the similar pages, we compute a patch using Xdelta [16]. The average patch size is 1,070 bytes, allowing them to be stored in 37,695 4-KB pages, saving 106,802 pages. In sum, sub-page sharing requires only 88,422 pages to store the memory for all VMs instead of 195,224 for full-page sharing or 393,120 originally—an impressive 77% savings, or almost another 50% over full-page sharing. We note that this was the least savings in our experiments; the savings from patching are even higher in most cases. Further, a significant amount of page sharing actually comes from zero pages and, therefore, depends on their availability. For instance, the same workload when executed on 256-MB VMs yields far fewer zero pages. Alternative mechanisms to page sharing become even more important in such cases.

Pages	Initial	After Sharing	After Patching
Unique	191,646	191,646	
Sharable (non-zero)	52,436	3,577	
Zero	149,038	1	
Total	393,120	195,224	88,422
Reference		50,727	50,727
Patchable		144,497	37,695

Table 1: Effectiveness of page sharing across three 512-MB VMs running Windows XP, Debian and Slackware Linux using 4-KB pages.

One of the principal complications with sub-page sharing is identifying candidate reference pages. Difference Engine uses a parameterized scheme to identify similar pages based upon the hashes of several 64-byte portions of each page. In particular, HashSimilarityDetector(k, s) hashes the contents of $(k \cdot s)$ 64-byte blocks at randomly chosen locations on the page, and then groups these hashes together into k groups of s hashes each. We use each group as an index into a hash table. In other words, higher values of s capture *local* similarity while higher k values incorporate *global* similarity. Hence, HashSimilarityDetector(1,1) will choose one block on a page and index that block; pages are considered similar if that block of data matches. HashSimilarityDetector(1,2) combines the hashes from two different locations in the page into one index of length two. HashSimilarityDetector(2,1) instead indexes each page twice: once based on the contents of a first block, and again based on the contents of a second block. Pages that match at least one of the two blocks are chosen as candidates. For each scheme, the number of candidates, c, specifies how many different pages the hash table tracks for each signature. With one candidate, we only store the first page found with each signature; for larger values, we keep multiple pages in the hash table for each index. When trying to build a patch, Difference Engine computes a patch between all matching pages and chooses the best one.

Figure 2 shows the effectiveness of this scheme for various parameter settings on the two workloads described above. On the X-axis, we have parameters in the format $(k, s), c$, and on the Y-axis we plot the total savings from patching *after* all identical pages have been

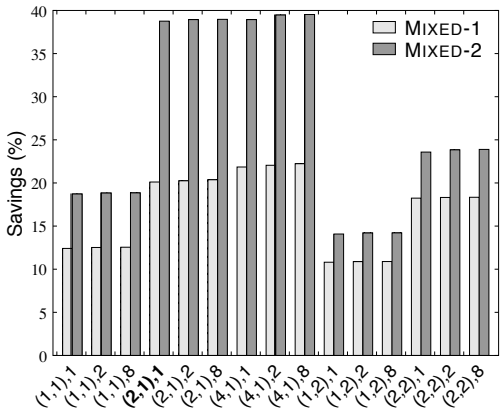


Figure 2: Effectiveness of the similarity detector for varying number of indices, index length and number of candidates. All entries use a 18-bit hash.

shared. Throughout the paper, we use the following definition of savings (we factor in the memory used to store the shared and patched/compressed pages):

$$\left(1 - \frac{\text{Total memory actually used}}{\text{Total memory allocated to VMs}}\right) \times 100$$

For both the workloads, HashSimilarityDetector(2,1) with one candidate does surprisingly well. There is a substantial gain due to hashing two distinct blocks in the page separately, but little additional gain by hashing more blocks. Combining blocks does not help much, at least for these workloads. Furthermore, storing more candidates in each hash bucket also produces little gain. Hence, Difference Engine indexes a page by hashing 64-byte blocks at two fixed locations in the page (chosen at random) and using each hash value as a separate index to store the page in the hash table. To find a candidate similar page, the system computes hashes at the same two locations, looks up those hash table entries, and chooses the better of the (at most) two pages found there.

Our current implementation uses 18-bit hashes to keep the hash table small to cope with the limited size of the Xen heap. In general though, larger hashes might be used for improved savings and fewer collisions. Our analysis indicates, however, that the benefits from increasing the hash size are modest. For example, using HashSimilarityDetector(2,1) with one candidate, a 32-bit hash yields a savings of 24.66% for MIXED-1, compared to a savings of 20.11% with 18-bit hashes.

3.3 Compression

Finally, for pages that are not significantly similar to other pages in memory, we consider compressing them to reduce the memory footprint. Compression is useful only if the compression ratio is reasonably high, and, like patching, if selected pages are accessed infrequently, otherwise the overhead of compression/decompression will outweigh the benefits. We identify candidate pages for compression using a global clock algorithm (Section 4.2), assuming that pages that have not been recently accessed are unlikely to be accessed in the near future.

Difference Engine supports multiple compression algorithms, currently LZO and WKdm as described in [25]; We invalidate compressed pages in the VM and save them in a dynamically allocated storage area in machine memory. When a VM accesses a compressed page, Difference Engine decompresses the page and returns it to the VM uncompressed. It remains there until it is again considered for compression.

3.4 Paging Machine Memory

While Difference Engine will deliver some (typically high) level of memory savings, in the worst case all VMs might actually require all of their allocated memory. Setting aside sufficient physical memory to account for this case prevents using the memory saved by Difference Engine to create additional VMs. Not doing so, however, may result in temporarily overshooting the physical memory capacity of the machine and cause a system crash. We therefore require a demand-paging mechanism to supplement main memory by writing pages out to disk in such cases.

A good candidate page for swapping out would likely not be accessed in the near future—the same requirement as compressed/patched pages. In fact, Difference Engine also considers compressed and patched pages as candidates for swapping out. Once the contents of the page are written to disk, the page can be reclaimed. When a VM accesses a swapped out page, Difference Engine fetches it from disk and copies the contents into a newly allocated page that is mapped appropriately in the VM's memory.

Since disk I/O is involved, swapping in/out is an expensive operation. Further, a swapped page is unavailable for sharing or as a reference page for patching. Therefore, swapping should be an infrequent operation. Difference Engine implements the core mechanisms for paging, and leaves policy decisions—such as when and how much to swap—to user space tools. We describe our reference implementation for swapping and the associated tools in Section 4.6.

4 Implementation

We have implemented Difference Engine on top of Xen 3.0.4 in roughly 14,500 lines of code. An additional 20,000 lines come from ports of existing patching and compression algorithms (Xdelta, LZO, WKdm) to run inside Xen.

4.1 Modifications to Xen

Xen and other platforms that support fully virtualized guests use a mechanism called "shadow page tables" to manage guest OS memory [24]. The guest OS has its own copy of the page table that it manages believing that they are the hardware page tables, though in reality it is just a map from the guest's virtual memory to its notion of physical memory (V2P map). In addition, Xen maintains a map from the guest's notion of physical memory to the machine memory (P2M map). The shadow page table is a cache of the results of composing the V2P map with the P2M map, mapping guest virtual memory directly to machine memory. Loosely, it is the virtualized analog to a software TLB. The shadow page table enables quick page translation and look-ups, and more importantly, can be used directly by the CPU.

Difference Engine relies on manipulating P2M maps and the shadow page tables to interpose on page accesses. For simplicity, we do not consider any pages mapped by Domain-0 (the privileged, control domain in Xen), which, among other things, avoids the potential for circular page faults. Our implementation method gives rise to two slight complications.

4.1.1 Real Mode

On x86 hardware, the booting-on-bare-metal process disables the x86 real-mode paging. This configuration is required because the OS needs to obtain some information from the BIOS for the boot sequence to proceed. When executing under Xen, this requirement means that paging is disabled during the initial stages of the boot process, and shadow page tables are not used until paging is turned on. Instead, the guest employs a direct P2M map as the page table. Hence, a VM's memory is not available for consideration by Difference Engine until paging has been turned on within the guest OS.

4.1.2 I/O Support

To support unmodified operating system requirements for I/O access, the Xen hypervisor must emulate much of the underlying hardware that the OS expects (such as the BIOS and the display device). Xen has a software I/O emulator based on Qemu [7]. A per-VM user-space process in Domain-0 known as `ioemu` performs all necessary I/O emulation. The `ioemu` must be able to read and write directly into the guest memory, primarily for efficiency. For instance, this enables the `ioemu` process to DMA directly into pages of the VM. By virtue of executing in Domain-0, the `ioemu` may map any pages of the guest OS in its address space.

By default, `ioemu` maps the entire memory of the guest into its address space for simplicity. Recall, however, that Difference Engine explicitly excludes pages mapped by Domain-0. Thus, `ioemu` will nominally prevent us from saving any memory at all, since every VM's address space will be mapped by its `ioemu` into Domain-0. Our initial prototype addressed this issue by modifying `ioemu` to map a small, fixed number (16) of pages from each VM at any given time. While simple to implement, this scheme suffered from the drawback that, for I/O-intensive workloads, the `ioemu` process would constantly have to map VM pages into its address space on demand, leading to undesirable performance degradation. To address this limitation, we implemented a dynamic aging mechanism in `ioemu`—VM pages are mapped into Domain-0 on demand, but not immediately unmapped. Every ten seconds, we unmap VM pages which were not accessed during the previous interval.

4.1.3 Block Allocator

Patching and compression may result in compact representations of a page that are much smaller than the page size. We wrote a custom block allocator for Difference Engine to efficiently manage storage for patched and compressed pages. The allocator acquires pages from the domain heap (from which memory for new VMs is allocated) on demand, and returns pages to the heap when no longer required.

4.2 Clock

Difference Engine implements a not-recently-used (NRU) policy [21] to select candidate pages for sharing, patching, compression and swapping out. On each invocation, the clock scans a portion of the memory, checking and clearing the *referenced* (R) and *modified* (M) bits on pages. Thus, pages with the R/M bits set must have been referenced/modified since the last scan. We ensure that successive scans of memory are separated by at least four seconds in the current implementation to give domains a chance to set the R/M bits on frequently accessed pages. In the presence of multiple VMs, the clock scans a small portion of each VM's memory in turn for fairness. The external API exported by the clock is simple: return a list of pages (of some maximum size) that have not been accessed in some time.

In OSes running on bare metal, the R/M bits on page-table entries are typically updated by the processor. Xen structures the P2M map exactly like the page tables used by the hardware. However, since the processor does not actually use the P2M map as a page table, the R/M bits are not updated automatically. We modify Xen's shadow page table code to set these bits when creating readable or writable page mappings. Unlike conventional operating systems, where there may be multiple sets of page tables that refer to the same set of pages, in Xen there is only one P2M map per domain. Hence, each guest page corresponds unambiguously to one P2M entry and one set of R/M bits.

Using the R/M bits, we can annotate each page with its "freshness":

- **Recently modified (C1):** The page has been written since the last scan. [M,R=1,1]
- **Not recently modified (C2):** The page has been accessed since the last scan, but not modified. [M,R=1,0]
- **Not recently accessed (C3):** The page has not been accessed at all since the last scan. [M,R=0,0]
- **Not accessed for an extended period (C4):** The page has not been accessed in the past few scans.

Note that the existing two R/M bits are not sufficient to classify C4 pages—we extend the clock's "memory" by leveraging two additional bits in the page table entries to identify such pages. We update these bits when a page is classified as C3 in consecutive scans. Together, these four annotations enable a clean separation between mechanism and policy, allowing us to explore different techniques to determine when and what to share, patch, and compress. By default, we employ the following policy. C1 pages are ignored; C2 pages are considered for sharing and to be reference pages for patching, but cannot be patched or compressed themselves; C3 pages can be shared or patched; C4 pages are eligible for everything, including compression and swapping.

We consider sharing first since it delivers the most memory savings in exchange for a small amount of meta data. We consider compression last because once a page is compressed, there is no opportunity to benefit from future sharing or patching of that page. An alternate, more aggressive policy might treat all pages as if they were in state C4 (not accessed in a long time)—in other words, proactively patch and compress pages. Initial experimentation indicates that while the contribution of patched and compressed pages does increase slightly, it does not yield a significant net savings. We also considered a policy that selects pages for compression before patching. Initial experimentation with the workloads in Section 5.4.2 shows that this policy performs slightly worse than the default in terms of savings, but incurs less performance overhead since patching is more resource intensive. We suspect that it may be a good candidate policy for heterogeneous workloads with infrequently changing working sets, but do not explore it further here.

4.3 Page Sharing

Difference Engine uses the SuperFastHash [4] function to compute digests for each scanned page and inserts them along with the page-frame number into a hash table. Ideally, the hash table should be sized so that it can hold entries for all of physical memory. The hash table is allocated out of Xen's heap space, which is quite limited in size: the code, data, and heap segments in Xen must all fit in a 12-MB region of memory. Changing the heap size requires pervasive code changes in Xen, and will likely break the application binary interface (ABI) for some OSes. We therefore restrict the size of the page-sharing hash table so that it can hold entries for only $1/5$ of physical memory. Hence Difference Engine processes memory in five passes, as described by Kloster *et al.* [13]. In our test configuration, this partitioning results in a 1.76-MB hash table. We divide the space of hash function values into five intervals, and only insert a page into the table if its hash value falls into the current interval. A complete cycle of five passes covering all the hash value intervals is required to identify all identical pages.

4.4 Page-similarity Detection

The goal of the page-similarity component is to find pairs of pages with similar content, and, hence, make candidates for patching. We implement a simple strategy for finding similar pages based on hashing short blocks within a page, as described in Section 3.2. Specifically, we use the HashSimilarityDetector(2,1) described there, which hashes short data blocks from two locations on each page, and indexes the page at each of those two locations in a separate page-similarity hash table, distinct from the page-sharing hash table described above. We use the 1-candidate variation, where at most one page is indexed for each block hash value.

Recall that the clock makes a complete scan through memory in five passes. The page-sharing hash table is cleared after each pass, since only pages *within* a pass are considered for sharing. However, two similar pages may appear in different passes if their hash values fall in different intervals. Since we want to only consider pages that have not been shared in a full cycle for patching, the page-similarity hash table is *not* cleared on every pass. This approach also increases the chances of finding better candidate pages to act as the reference for a patch.

The page-similarity hash table *may* be cleared after considering every page in memory—that is, at the end of each cycle of the global clock. We do so to prevent stale data from accumulating: if a page changes after it has been indexed, we should remove old pointers to it. Since we do not trap on write operations, it is simpler to just discard and rebuild the similarity hash table.

Only the last step of patching—building the patch and replacing the page with it—requires a lock. We perform all earlier steps (indexing and lookups to find similar pages) without pausing any domains. Thus, the page contents may change after Difference Engine indexes the page, or after it makes an initial estimate of patch size. This is fine since the goal of these steps is to find pairs of pages that will likely patch well. An intervening page

modification will not cause a correctness problem, only a patch that is larger than originally intended.

4.5 Compression

Compression operates similarly to patching—in both cases the goal is to replace a page with a shorter representation of the same data. The primary difference is that patching makes use of a reference page, while a compressed representation is self contained.

There is one important interaction between compression and patching: once we compress a page, the page can no longer be used as a reference for a later patched page. A naive implementation that compresses all non-identical pages as it goes along will almost entirely prevent page patches from being built. Compression of a page should be postponed at least until all pages have been checked for similarity against it. A complete cycle of a page sharing scan will identify similar pages, so a sufficient condition for compression is that *no page should be compressed until a complete cycle of the page sharing code finishes*. We make the definition of "not accessed for an extended period" in the clock algorithm coincide with this condition (state C4). As mentioned in Section 4.2, this is our default policy for page compression.

4.6 Paging Machine Memory

Recall that any memory freed by Difference Engine cannot be used reliably without supplementing main memory with secondary storage. That is, when the total allocated memory of all VMs exceeds the system memory capacity, some pages will have to be swapped to disk. Note that this ability to overcommit memory is useful in Xen independent of other Difference Engine functionality, and has been designed accordingly.

The Xen VMM does not perform any I/O (delegating all I/O to Domain-0) and is not aware of any devices. Thus, it is not possible to build swap support directly in the VMM. Further, since Difference Engine supports unmodified OSes, we cannot expect any support from the guest OS. Figure 3 shows the design of our swap implementation guided by these constraints. A single swap daemon (`swapd`) running as a user process in Domain-0 manages the swap space. For each VM in the system, `swapd` creates a separate thread to handle swap-in requests. Swapping out is initiated by `swapd`, when one of the following occurs:

- The memory utilization in the system exceeds some user configurable threshold (the `HIGH_WATERMARK`). Pages are swapped out until a user configurable threshold of free memory is attained (the `LOW_WATERMARK`). A separate thread (the `memory_monitor`) tracks system memory.

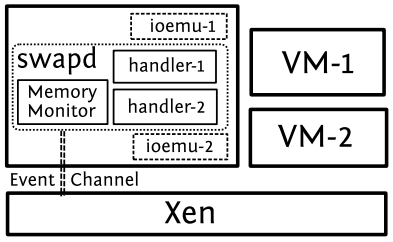


Figure 3: Architecture of the swap mechanism.

- A swap-out notification is received from Xen via an event channel. This allows the hypervisor to initiate swapping if more memory is urgently required (for instance, when creating a private copy of a shared page). The hypervisor indicates the amount of free memory desired.

- A swap-out request is received from another process. This allows other user space tools (for instance, the VM creation tool) to initiate swapping to free memory. We currently employ XenStore [5] for such communication, but any other IPC mechanism can be used.

Note that `swapd` always treats a swap-out request as a hint. It will try to free pages, but if that is not possible—if no suitable candidate page was available, for instance, or if the swap space became full—it continues silently. A single flat file of configurable size is used as storage for the swap space.

To swap out a page, `swapd` makes a hypercall into Xen, where a victim page is chosen by invoking the global clock. If the victim is a compressed or patched page, we first reconstruct it. We pause the VM that owns the page and copy the contents of the page to a page in Domain-0's address space (supplied by `swapd`). Next, we remove all entries pointing to the victim page in the P2M and M2P maps, and in the shadow page tables. We then mark the page as swapped out in the corresponding page table entry. Meanwhile, `swapd` writes the page contents to the swap file and inserts the corresponding byte offset in a hash table keyed by <Domain ID, guest page-frame number>. Finally, we free the page, return it to the domain heap, and reschedule the VM.

When a VM tries to access a swapped page, it incurs a page fault and traps into Xen. We pause the VM and allocate a fresh page to hold the swapped in data. We populate the P2M and M2P maps appropriately to accommodate the new page. Xen dispatches a swap-in request to `swapd` containing the domain ID and the faulting page-frame number. The handler thread for the faulting domain in `swapd` receives the request and fetches the location of the page in the swap file from the hash table. It then copies the page contents into the newly allocated page frame within Xen via another hypercall. At

Function	Mean execution time (μs)
share_pages	6.2
cow_break	25.1
compress_page	29.7
uncompress	10.4
patch_page	338.1
unpatch	18.6
swap_out_page	48.9
swap_in_page	7151.6

Table 2: CPU overhead of different functions.

this point, swapd notifies Xen, and Xen restarts the VM at the faulting instruction.

This implementation leads to two interesting interactions between ioemu and swapd. First, recall that ioemu can directly write to a VM page mapped in its address space. Mapped pages might not be accessed until later, so a swapped page can get mapped or a mapped page can get swapped out without immediate detection. To avoid unnecessary subsequent swap ins, we modify ioemu to ensure that pages to be mapped will be first swapped in if necessary and that mapped pages become ineligible for swapping. Also note that control *must* be transferred from Xen to swapd for a swap in to complete. This asynchrony allows a race condition where ioemu tries to map a swapped out page (so Xen initiates a swap in on its behest) and proceeds with the access before the swap in has finished. This race can happen because both processes must run in Domain-0 in the Xen architecture. As a work around, we modify ioemu to block if a swap in is still in progress inside swapd using shared memory between the processes for the required synchronization.

5 Evaluation

We first present micro-benchmarks to evaluate the cost of individual operations, the performance of the global clock and the behavior of each of the three mechanisms in isolation. Next, we evaluate whole system performance: for a range of workloads, we measure memory savings and the impact on application performance. We quantify the contributions of each Difference Engine mechanism, and also present head-to-head comparisons with the VMware ESX server. Finally, we demonstrate how our memory savings can be used to boost the aggregate system performance. Unless otherwise mentioned, all experiments are run on dual-processor, dual-core 2.33-GHz Intel Xeon machines and the page size is 4 KB.

5.1 Cost of Individual Operations

Before quantifying the memory savings provide by Difference Engine, we measure the overhead of various functions involved. We obtain these numbers by enabling each mechanism in isolation, and running the custom micro-benchmark described in Section 5.3. To benchmark paging, we disabled all three mechanisms and forced eviction of 10,000 pages from a single 512-MB VM. We then ran a simple program in the VM that touches all memory to force pages to be swapped in.

Table 2 shows the overhead imposed by the major Difference Engine operations. As expected, collapsing identical pages into a copy-on-write shared page (share_page) and recreating private copies (cow_break) are relatively cheap operations, taking approximately 6 and 25 μs, respectively. Perhaps more surprising, however, is that compressing a page on our hardware is fast, requiring slightly less than 30 μs on average. Patching, on the other hand, is almost an order of magnitude slower: creating a patch (patch_page) takes over 300 μs. This time is primarily due to the overhead of finding a good candidate base page and constructing the patch. Both decompressing a page and re-constructing a patched page are also fairly fast, taking 10 and 18 μs respectively.

Swapping out takes approximately 50 μs. However, this does *not* include the time to actually write the page to disk. This is intentional: once the page contents have been copied to user space, they *are* immediately available for being swapped in; and the actual write to the disk might be delayed because of file system and OS buffering in Domain-0. Swapping in, on the other hand, is the most expensive operation, taking approximately 7 ms. There are a few caveats, however. First, swapping in is an asynchronous operation and might be affected by several factors, including process scheduling within Domain-0; it is *not* a tight bound. Second, swapping in might require reading the page from disk, and the seek time will depend on the size of the swap file, among other things.

5.2 Clock Performance

The performance of applications running with Difference Engine depends upon how effectively we choose idle pages to compress or patch. Patching and compression are computationally intensive, and the benefits of this overhead last only until the next access to the page. Reads are free for shared pages, but not so for compressed or patched pages. The clock algorithm is intended to only consider pages for compression/patching that are not likely to be accessed again soon; here we evaluate how well it achieves that goal.

For three different workloads, we trace the lifetime of each patched and compressed page. The lifetime of a page is the time between when it was patched/compressed, and the time of the first subsequent access (read or write). The workloads range from best case homogeneous configurations (same OS, same applications) to a worst case, highly heterogeneous mix (different OSes, different applications). The RUBiS and

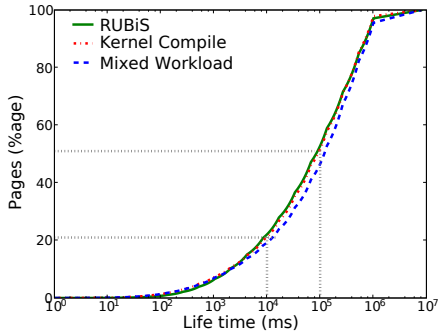


Figure 4: Lifetime of patched and compressed pages for three different workloads. Our NRU implementation works well in practice.

kernel compile workloads use four VMs each (Section 5.4.1). We use the MIXED-1 workload described earlier (Section 3.2) as the heterogeneous workload.

Figure 4 plots the cumulative distribution of the lifetime of a page: the X-axis shows the lifetime (in ms) in log scale, and the Y-axis shows the fraction of compressed/patched pages. A good clock algorithm should give us high lifetimes, since we would like to patch/compress only those pages which will not be accessed in the near future. As the figure shows, almost 80% of the victim pages have a lifetime of at least 10 seconds, and roughly 50% have a lifetime greater than 100 seconds. This is true for both the homogeneous and the mixed workloads, indicating that our NRU implementation works well in practice.

5.3 Techniques in Isolation

To understand the individual contribution of the three techniques, we first quantify the performance of each in isolation. We deployed Difference Engine on three machines running Debian 3.1 on a VM. Each machine is configured to use a single mechanism—one machine uses just page sharing, one uses just compression, and one just patching. We then subject all the machines to the same workload and profile the memory utilization.

To help distinguish the applicability of each technique to various page contents, we choose a custom workload generator that manipulates memory in a repeatable, predictable manner over off-the-shelf benchmarks. Our workload generator runs in four phases. First it allocates pages of a certain type. To exercise the different mechanisms in predictable ways, we consider four distinct page types: zero pages, random pages, identical pages and similar-but-not-identical pages. Second, it reads all the allocated pages. Third, it makes several small writes to all the pages. Finally, it frees all allocated pages and exits. After each step, the workload generator idles for some time, allowing the memory to stabilize. For each run of the benchmark, we spawn a new VM and start the workload generator within it. At the end of each run, we destroy the container VM and again give memory some time to stabilize before the next run. We ran benchmarks with varying degrees of similarity, where similarity is defined as follows: a similarity of 90% means all pages differ from a base page by 10%, and so any two pages will differ from each other by at most 20%. Here, we present the results for 95%-similar pages, but the results for other values are similar.

Each VM image is configured with 256 MB of memory. Our workload generator allocates pages filling 75% (192 MB) of the VM's memory. The stabilization period is a function of several factors, particularly the period of the global clock. For these experiments, we used a sleep time of 80 seconds between each phase. During the write step, the workload generator writes a single constant byte at 16 fixed offsets in the page. On each of the time series graphs, the significant events during the run are marked with a vertical line. These events are: (1) begin and (2) end of the allocation phase, (3) begin and (4) end of the read phase, (5) begin and (6) end of the write phase, (7) begin and (8) end of the free phase, and (9) VM destruction.

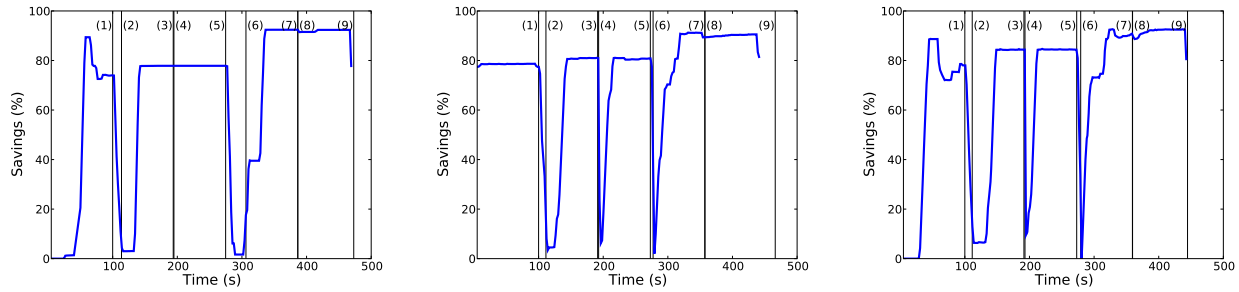
Figure 5 shows the memory savings as a function of time for each mechanism for identical pages (for brevity, we omit results with zero pages—they are essentially the same as identical pages). Note that while each mechanism achieves similar savings, the crucial difference is that reads are free for page sharing. With compression/patching, even a read requires the page to be reconstructed, leading to the sharp decline in savings around event (3) and (5).

At the other extreme are random pages. Intuitively, none of the mechanisms should work well since the opportunity to share memory is scarce. Figure 6 agrees: once the pages have been allocated, none of the mechanisms are able to share more than 15–20% memory. Page sharing does the worst, managing 5% at best.

From the perspective of page sharing, similar pages are no better than random pages. However, patching should take advantage of sub-page similarity across pages. Figure 7 shows the memory savings for the workload with pages of 95% similarity. Note how similar the graphs for sharing and compression look for similar and random pages. Patching, on the other hand, does substantially better, extracting up to 55% savings.

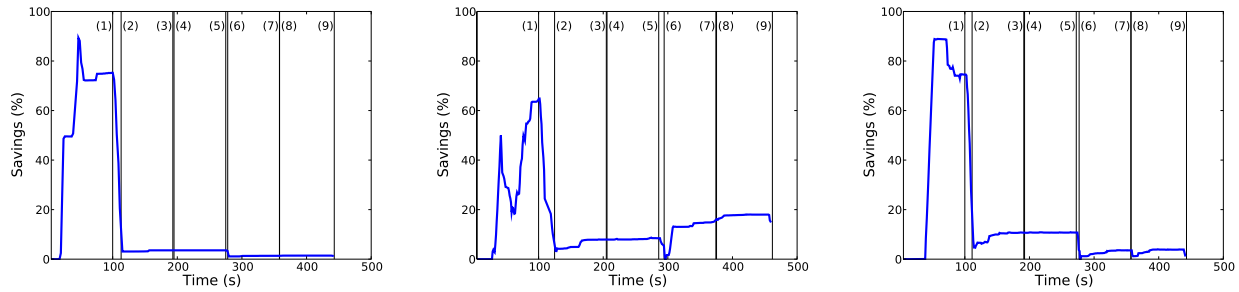
5.4 Real-world Applications

We now present the performance of Difference Engine on a variety of workloads. We seek to answer two questions. First, how effective are the memory-saving mechanisms at reducing memory usage for real-world applications? Second, what is the impact of those memory-sharing mechanisms on system performance? Since the



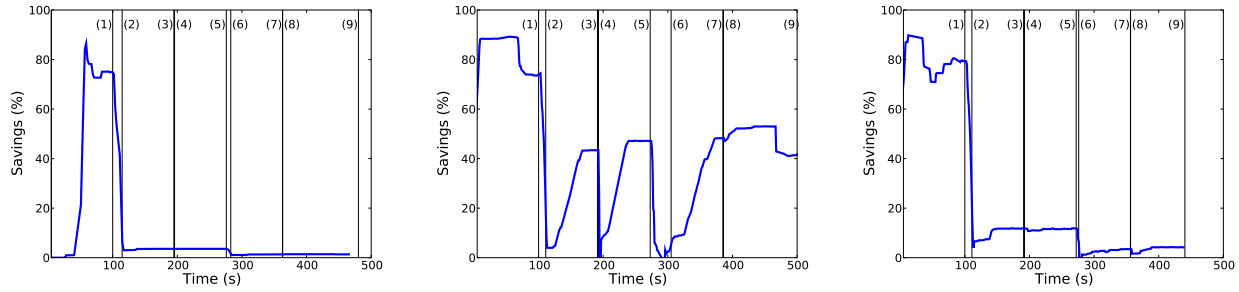
(a) Sharing (b) Patching (c) Compression

Figure 5: Workload: Identical Pages. Performance with zero pages is very similar. All mechanisms exhibit similar gains.



(a) Sharing (b) Patching (c) Compression

Figure 6: Workload: Random Pages. None of the mechanisms perform very well, with sharing saving the least memory.



(a) Sharing (b) Patching (c) Compression

Figure 7: Workload: Similar Pages with 95% similarity. Patching does significantly better than compression and sharing.

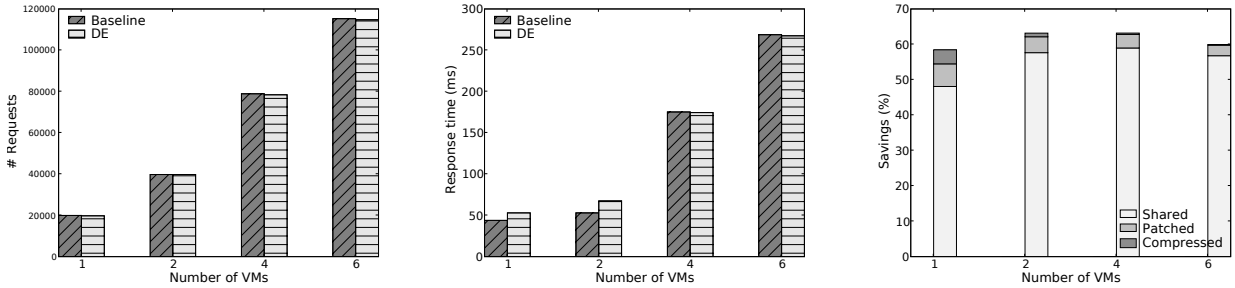
degree of possible sharing depends on the software configuration, we consider several different cases of application mixes.

To put our numbers in perspective, we conduct head-to-head comparisons with VMware ESX server for three different workload mixes. We run ESX Server 3.0.1 build 32039 on a Dell PowerEdge 1950 system. Note that even though this system has two 2.3-GHz Intel Xeon processors, our VMware license limits our usage to a single CPU. We therefore restrict Xen (and, hence, Difference Engine) to use a single CPU for fairness. We also ensure that the OS images used with ESX match those used with Xen, especially the file system and disk layout. Note that we are only concerned with the effectiveness of the memory sharing mechanisms—not in comparing the application performance across the two hypervisors. Further, we configure ESX to use its most aggressive page sharing settings where it scans 10,000 pages/second (default 200); we configure Difference Engine similarly.

5.4.1 Base Scenario: Homogeneous VMs

In our first set of benchmarks, we test the base scenario where all VMs on a machine run the same OS and applications. This scenario is common in cluster-based systems where several services are replicated to provide fault tolerance or load balancing. Our expectation is that significant memory savings are available and that most of the savings will come from page sharing.

On a machine running standard Xen, we start from 1 to 6 VMs, each with 256 MB of memory and running RUBiS [10]—an e-commerce application designed to evaluate application server performance—on Debian



(a) Total requests handled (b) Average response time (c) Average and maximum savings

Figure 8: Difference Engine performance with homogeneous VMs running RUBiS

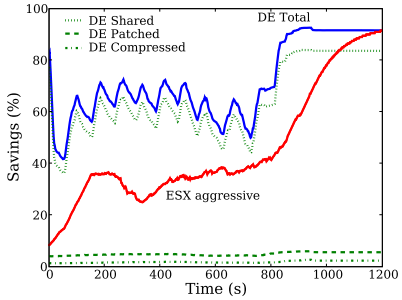


Figure 9: Four identical VMs execute dbench. For such homogeneous workloads, both Difference Engine and ESX eventually yield similar savings, but DE extracts more savings *while* the benchmark is in progress.

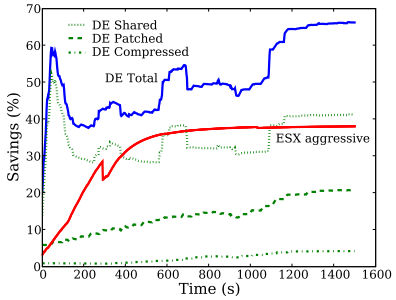


Figure 10: Memory savings for MIXED-1. Difference Engine saves up to 45% more memory than ESX.

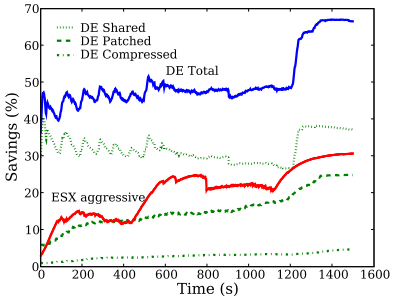


Figure 11: Memory savings for MIXED-2. Difference Engine saves almost twice as much memory as ESX.

3.1. We use the PHP implementation of RUBiS; each instance consists of a Web server (Apache) and a database server (MySQL). Two distinct client machines generate the workload, each running the standard RUBiS workload generator simulating 100 user sessions. The benchmark runs for roughly 20 minutes. The workload generator reports several metrics at the end of the benchmark, in particular the average response time and the total number of requests served.

We then run the same set of VMs with Difference Engine enabled. Figures 8(a) and 8(b) show that both the total number of requests and the average response time remain unaffected while delivering 65–75% memory savings in all cases. In Figure 8(c), the bars indicate the average memory savings over the duration of the benchmark. Each bar also shows the individual contribution of each mechanism. Note that in this case, the bulk of memory savings comes from page sharing. Recall that Difference Engine tries to share as many pages as it can before considering pages for patching and compression, so sharing is expected to be the largest contributor in most cases, particularly in homogeneous workloads.

Next, we conduct a similar experiment where each VM compiles the Linux kernel (version 2.6.18). Since the working set of VMs changes much more rapidly in a kernel compile, we expect less memory savings compared to the RUBiS workload. As before, we measure the time taken to finish the compile and the memory savings for varying number of virtual machines. We summarize the results here for brevity: in each case, the performance under Difference Engine is within 5% of the baseline, and on average Difference Engine delivers around 40% savings with four or more VMs.

We next compare Difference Engine performance with the VMware ESX server. We set up four 512-MB virtual machines running Debian 3.1. Each VM executes dbench [2] for ten minutes followed by a stabilization period of 20 minutes. Figure 9 shows the amount of memory saved as a function of time. First, note that *eventually* both ESX and Difference Engine reclaim roughly the same amount of memory (the graph for ESX plateaus beyond 1,200 seconds). However, *while* dbench is executing, Difference Engine delivers approximately 1.5 times the memory savings achieved by ESX. As before, the bulk of Difference Engine savings come from page sharing for the homogeneous workload case.

5.4.2 Heterogeneous OS and Applications

Given the increasing trend towards virtualization, both on the desktop and in the data center, we envision that a single physical machine will host significantly different types of operating systems and workloads. While smarter

	Kernel Compile (sec)	Vim compile, lmbench (sec)	RUBiS requests	RUBiS response time(ms)
Baseline	670	620	3149	1280
DE	710	702	3130	1268

Table 3: Application performance under Difference Engine for the heterogeneous workload MIXED-1 is within 7% of the baseline.

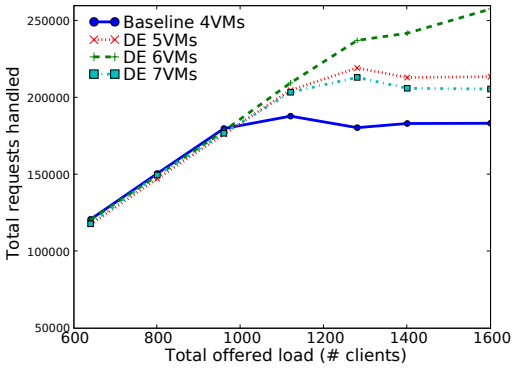
VM placement and scheduling will mitigate some of these differences, there will still be a diverse and heterogeneous mix of applications and environments, underscoring the need for mechanisms other than page sharing. We now examine the utility of Difference Engine in such scenarios, and demonstrate that significant additional memory savings result from employing patching and compression in these settings.

Figures 10 and 11 show the memory savings as a function of time for the two heterogeneous workloads—MIXED-1 and MIXED-2 described in Section 3.2. We make the following observations. First, in steady state, Difference Engine delivers a factor of 1.6-2.5 more memory savings than ESX. For instance, for the MIXED-2 workload, Difference Engine could host the three VMs allocated 512 MB of physical memory each in approximately 760 MB of machine memory; ESX would require roughly 1100 MB of machine memory. The remaining, significant, savings come from patching and compression. And these savings come at a small cost. Table 3 summarizes the performance of the three benchmarks in the MIXED-1 workload. The baseline configuration is regular Xen without Difference Engine. In all cases, performance overhead of Difference Engine is within 7% of the baseline. For the same workload, we find that performance under ESX with aggressive page sharing is also within 5% of the ESX baseline with no page sharing.

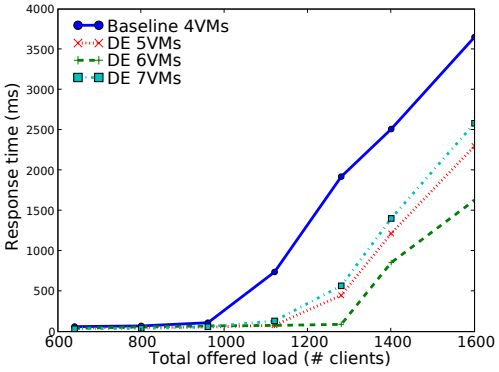
5.4.3 Increasing Aggregate System Performance

Difference Engine goes to great lengths to reclaim memory in a system, but eventually this extra memory needs to actually get used in a productive manner. One can certainly use the saved memory to create more VMs, but does that increase the aggregate system performance?

To answer this question, we created four VMs with 650 MB of RAM each on a physical machine with 2.8 GB of free memory (excluding memory allocated to Domain-0). For the baseline (without Difference Engine), Xen allocates memory statically. Upon creating all the VMs, there is clearly not enough memory left to create another VM of the same configuration. Each VM hosts a RUBiS instance. For this experiment, we used the Java Servlets implementation of RUBiS. There are two distinct client machines per VM to act as workload generators.



(a) Total requests handled



(b) Average response time

Figure 12: Up to a limit, Difference Engine can help increase aggregate system performance by spreading the load across extra VMs.

The goal is to increase the load on the system to saturation. The solid lines in Figures 12(a) and 12(b) show the total requests served and the average response time for the baseline, with the total offered load marked on the X-axis. Note that beyond 960 clients, the total number of requests served plateaus at around 180,000 while the average response time increases sharply. Upon investigation, we find that for higher loads all of the VMs have more than 95% memory utilization and some VMs actually start swapping to disk (within the guest OS). Using fewer VMs with more memory (for example, 2 VMs with 1.2 GB RAM each) did not improve the baseline performance for this workload.

Next, we repeat the same experiment with Difference Engine, except this time we utilize reclaimed memory to create additional VMs. As a result, for each data point on the X-axis, the per VM load decreases, while the aggregate offered load remains the same. We expect that since each VM individually has lower load compared to the baseline, the system will deliver better aggregate performance. The remaining lines in Figures 12(a) and 12(b) show the performance with up to three extra VMs. Clearly, Difference Engine enables higher aggregate performance and better response time compared to the baseline. However, beyond a certain point (two additional

VMs in this case), the overhead of managing the extra VMs begins to offset the performance benefits: Difference Engine has to effectively manage 4.5 GB of memory on a system with 2.8 GB of RAM to support seven VMs. In each case, beyond 1400 clients, the VMs working set becomes large enough to invoke the paging mechanism: we observe between 5,000 pages (for one extra VM) to around 20,000 pages (for three extra VMs) being swapped out, of which roughly a fourth get swapped back in.

6 Conclusion

One of the primary bottlenecks to higher degrees of virtual machine multiplexing is main memory. Earlier work shows that substantial memory savings are available from harvesting identical pages across virtual machines when running homogeneous workloads. The premise of this work is that there are significant additional memory savings available from locating and patching similar pages and in-memory page compression. We present the design and evaluation of Difference Engine to demonstrate the potential memory savings available from leveraging a combination of whole page sharing, page patching, and compression. We discuss our experience addressing a number of technical challenges, including: i) algorithms to quickly identify candidate pages for patching, ii) demand paging to support over-subscription of total assigned physical memory, and iii) a clock mechanism to identify appropriate target machine pages for sharing, patching, compression and paging. Our performance evaluation shows that Difference Engine delivers an additional factor of 1.6–2.5 more memory savings than VMware ESX Server for a variety of workloads, with minimal performance overhead. Difference Engine mechanisms might also be leveraged to improve single OS memory management; we leave such exploration to future work.

References

[1] `http://sysbench.sourceforge.net/`.

[2] `http://samba.org/ftp/tridge/dbench/`.

[3] `http://www.iozone.org/`.

[4] `http://www.azillionmonkeys.com/qed/hash.html`.

[5] `http://wiki.xensource.com/xenwiki/XenStore`.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[7] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*, 2005.

[8] A. Z. Broder. Identifying and filtering near-duplicate documents. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, 2000.

[9] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, 1997.

[10] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of EJB applications. In *Proceedings of the 17th ACM Conference on Object-oriented programming, systems, languages, and applications*, 2002.

[11] F. Douglis. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of the USENIX Winter Technical Conference*, 1993.

[12] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the USENIX Annual Technical Conference*, 2003.

[13] J. F. Kloster, J. Kristensen, and A. Mejlholm. On the feasibility of memory sharing. Master's thesis, Aalborg University, 2006.

[14] P. Kulkarni, F. Douglis, J. Lavoie, and J. M. Tracey. Redundancy elimination within large collections of files. In *Proceedings of the USENIX Annual Technical Conference*, 2004.

[15] S. Low. Connectix RAM doubler information. `http://www.lowtek.com/maxram/rd.html`, May 1996.

[16] J. MacDonald. xdelta. `http://www.xdelta.org/`.

[17] V. Makhija, B. Herndon, P. Smith, L. Roderick, E. Zamost, and J. Anderson. VMmark: A scalable benchmark for virtualized systems. Technical Report TR 2006-002, VMware, 2006.

[18] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proceedings of the USENIX Winter Technical Conference*, 1994.

[19] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX Annual Technical Conference*, 1996.

[20] K.-T. Moeller. Virtual machine benchmarking. Diploma thesis, System Architecture Group, University of Karlsruhe, Germany, 2007.

[21] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2007.

[22] I. C. Tuduce and T. Gross. Adaptive main memory compression. In *Proceedings of the USENIX Annual Technical Conference*, 2005.

[23] M. Vrable, J. Ma, J. Chen, D. Moore, E. VandeKieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity and containment in the Potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating System Principles*, 2005.

[24] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th ACM/USENIX Symposium on Operating System Design and Implementation*, 2002.

[25] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of the USENIX Annual Technical Conference*, 1999.

Quanto: Tracking Energy in Networked Embedded Systems

Rodrigo Fonseca[†⋆], *Prabal Dutta*[†], *Philip Levis*[‡], *and Ion Stoica*[†]
{rfonseca,prabal,istoica}@cs.berkeley.edu {pal}@cs.stanford.edu

[†]*Computer Science Division*
University of California, Berkeley
Berkeley, CA

[⋆]*Yahoo! Research*
Santa Clara, CA

[‡]*Computer Systems Laboratory*
Stanford University
Stanford, CA

Abstract

We present Quanto, a network-wide time and energy profiler for embedded network devices. By combining well-defined interfaces for hardware power states, fast high-resolution energy metering, and causal tracking of programmer-defined activities, Quanto can map how energy and time are spent on nodes and across a network. Implementing Quanto on the TinyOS operating system required modifying under 350 lines of code and adding 1275 new lines. We show that being able to take fine-grained energy consumption measurements as fast as reading a counter allows developers to precisely quantify the effects of low-level system implementation decisions, such as using DMA versus direct bus operations, or the effect of external interference on the power draw of a low duty-cycle radio. Finally, Quanto is lightweight enough that it has a minimal effect on system behavior: each sample takes 100 CPU cycles and 12 bytes of RAM.

1 Introduction

Energy is a scarce resource in embedded, battery-operated systems such as sensor networks. This scarcity has motivated research into new system architectures [16], platform designs [17], medium access control protocols [36], networking abstractions [26], transport layers [23], operating system abstractions [21], middleware protocols [11], and data aggregation services [22]. In practice, however, the energy consumption of deployed systems differs greatly from expectations or what lab tests suggest. In one network designed to monitor the microclimate of redwood trees, for example, 15% of the nodes died after one week, while the rest lasted for months [33]. The deployers of the network hypothesize that environmental conditions – poor radio connectivity, leading to time synchronization failure – caused the early demise of these nodes, but a lack of data makes the exact cause unknown.

Understanding how and why an embedded application spends energy requires answering numerous questions. For example, how much energy do individual operations, such as sampling sensors, receiving packets, or using CPU, cost? What is the energy breakdown of a node, in terms of activity, hardware, and time? Network-wide, how much energy do network services such as routing, time synchronization, and localization, consume?

Three factors make these questions difficult to answer. First, nodes need to be able to measure the actual draw of their hardware components.While software models of system energy are reasonably accurate in controlled environments, networks in the wild often experience externalities, such as 100°C temperature swings [32], electrical shorts due to condensation [31], and 802.11 interference [24]. Second, nodes have limited storage capability, on the order of kilobytes of RAM, and profile collection must be very lightweight, so it is energy efficient and minimizes its effect on system behavior. Finally, there is a semantic gap between common abstractions, such as threads or subsystems, and the actual entities a developer cares about for resource accounting. This gap requires a profiling system to tie together separate operations across multiple energy consumers, such as sampling sensors, sending packets, and CPU operations.

This paper presents Quanto, a time and energy profiler that addresses these challenges through four research contributions. First, we leverage an energy sensor based on a simple switching regulator [9] to enable an OS to take fine-grained measurements of energy usage as cheaply as reading a counter. Second, we show that a post-facto regression can distinguish the energy draw of individual hardware components, thereby only requiring the OS to sample aggregate system consumption. Third, we describe a simple labeling mechanism that causally connects this energy usage to high-level, programmer-defined activities. Finally, we extend these techniques to track network-wide energy usage in terms of node-local actions. We briefly outline these ideas next.

Energy in an embedded system is spent by a set of hardware components operating concurrently, responding to application and external events. As a first step in understanding energy usage, Quanto determines the energy breakdown by hardware component over time. A system has several hardware components, like the CPU, radio, and flash memory, and each one has different functional units, which we call *energy sinks*. Each energy sink has operating modes with different power draws, which we call *power states*. At any given time, the aggregate power draw for a system is determined by the set of active power states of its energy sinks.

In many embedded systems, the system software can closely track the hardware components' power states and state transitions. We modify device drivers to track and expose hardware power states to the OS in real-time. The OS combines this information with fine-grained, timely measurements of system-wide energy usage taken using a high-resolution, low-latency energy meter. Every time any hardware component changes its power state, the OS records how much energy was used, and how much time has passed since the immediately preceding power state change. For each interval during which the power states are constant, this generates one equation relating the active power states, the energy used, the time spent, and the unknown power draw of a particular energy sink's power state. Over time, a family of equations are generated and can be solved (i.e. the power draw of individual energy sinks can be estimated) using multivariate linear regression. Section 2 presents the details of this approach.

The next step is to tie together the energy used by different hardware components on behalf of high-level activities such as sensing, routing, or computing, for which we need an abstraction at the appropriate granularity. Earlier work has profiled energy usage at the level of instructions [8], performance events [7], program counter [13], procedures [14], processes [30], and software modules [28]. In this work, we borrow the *activity* abstraction of a resource principal [2, 19]. An activity is a logical set of operations whose resource usage should be grouped together. In the embedded systems we consider, it is essential that activities span hardware components other than the CPU, and even different nodes.

To account for the resource consumption of activities, Quanto tracks when a hardware component is performing operations on behalf of an activity. Each activity is given a label, and the OS propagates this label to all causally related operations. As an analogy, this tracking is accomplished by conceptually "painting" a hardware component the same "color" as the activity for which it is doing work. To transfer activity labels across nodes, Quanto inserts a field in each packet that includes the initiating activity's label. More specifically, when a packet is passed to the network stack for transmission, the packet's activity field is set to the CPU's then-current activity. This ensures that a transmitted packet is labeled the same as the activity which initiated its submission. Upon reception, Quanto reads the packet's hidden field and sets the CPU activity to the activity noted in the packet. In Section 3 we give a more precise definition of activities, and detail how Quanto implements activity tracking across both the hardware components of a single node and across the nodes in a network.

The final step is to merge this information. Quanto records events for both power state and activity changes for each hardware resource. In our current prototype, we use these logs to perform this step *post-facto*. From the power states log and the regression, we know the active power state and the power draw for each hardware component; from the activities log, we know on behalf of which activity the component was doing work. Combining these two pieces of information provides a thorough breakdown of energy consumption over time.

To evaluate the functionality and performance of Quanto, we implemented the framework in TinyOS, a popular sensornet operating system. Implementing our approach required small changes to six OS abstractions – timers, tasks, arbiters, network stack, interrupt handlers, and device drivers. We changed 22 files and 171 lines of code for core OS primitives, and 16 files and 148 lines of code for representative device drivers, to support activity tracing and exposing of power states.

2 Energy Tracking

In this section, we present how Quanto answers the question *where have all the joules gone?* This requires distinguishing the individual energy consumption of hardware components or peripherals that are operating concurrently when only their aggregate energy usage is observable. We ground our discussion on the specific hardware and software platform in our prototype, although we believe the techniques to be applicable to other platforms as well. We briefly sketch our approach in the next subsection and then use the remainder of this section to describe Quanto's energy tracking framework in detail.

2.1 Overview

In many embedded systems, the OS can track the power states and state transitions of the platform's various energy sinks. This power state information can be combined with snapshots of the aggregate energy consumption to infer the consumption of individual sinks.

We call each functional unit in a system an *energy sink*, and their different operating modes *power states*. Quanto modifies the device drivers to intercept all events which change the power state of an energy sink. The OS itself

keeps track of both the energy usage, ΔE, and elapsed time, Δt, between any two such events. Since the OS tracks the active sinks and their power states, it is able to generate one linear equation of the following form for each interval

$$\Delta E = \Delta t \sum_{i=0}^{n} \alpha_i p_i \quad (1)$$

where the average power over the interval, P, is $\Delta E / \Delta t$. The variable α_i is a binary variable indicating whether the i-th power state was active during the interval, and p_i is the (unknown) power draw of the i-th state. The limit n represents the total number of power states over all energy sinks in the system. In one time interval, this equation is not solvable (unless only one power state is active), but over time, an application generates a system of equations as different energy sinks transition through different power states. When the system of equations is sufficiently constrained, a simple linear regression yields the individual power draws.

2.2 Hardware Platform

Because it samples the accumulated energy consumption at every hardware power state change, Quanto requires high-resolution, low-latency, and low-overhead energy measurements. These readings must closely reflect the energy consumed during the preceding interval. To accomplish this, our implementation uses the iCount energy meter [9]. The iCount implementation on this platform exhibits a maximum error of $\pm 15\%$ over five orders of magnitude in current draw, an energy resolution of approximately 1 μJ, a read latency of 24 μs (24 instruction cycles), and a power overhead that ranges from 1% when the node is in standby to 0.01% when the node is active, for a typical workload.

We used the HydroWatch platform which incorporates iCount into a custom sensornet node [10]. This platform uses the Texas Instruments 16-bit MSP430F1611 microcontroller with 48 KB of internal flash memory and 10 KB of RAM, an 802.15.4-compliant CC2420 radio, and an Atmel 16-Mbit AT45DB161D NOR flash memory. The platform also includes three LEDs.

2.3 Energy Sinks and Power States

The Hydrowatch platform's energy sinks, and their nominal current draws, are shown in Table 1. The microcontroller includes several different functional units. The microcontroller's eight energy sinks have sixteen power states but since many of the energy sinks can operate independently, the microcontroller can exhibit hundreds of distinct draw profiles. The five energy sinks in the radio have fourteen power states. Some of these states are mutually exclusive. For example, the radio cannot use both receive and transmit at the same time. Similarly, the flash memory can operate in several distinct power states. Collectively, the energy sinks represented by the microcontroller, radio, flash memory, and LEDs can operate independently, so, in principle, the system may exhibit hundreds or thousands of distinct power profiles.

Energy Sink	Power State	Current
Microcontroller		
CPU	ACTIVE	500 μA
	LPM0	75 μA
	LPM1†	75 μA
	LPM2	17 μA
	LPM3	2.6 μA
	LPM4	0.2 μA
Voltage Reference	ON	500 μA
ADC	CONVERTING	800 μA
DAC	CONVERTING-2	50 μA
	CONVERTING-5	200 μA
	CONVERTING-7	700 μA
Internal Flash	PROGRAM	3 mA
	ERASE	3 mA
Temperature Sensor	SAMPLE	60 μA
Analog Comparator	COMPARE	45 μA
Supply Supervisor	ON	15 μA
Radio		
Regulator	OFF	1 μA
	ON	22 μA
	POWER_DOWN	20 μA
Batter Monitor	ENABLED	30 μA
Control Path	IDLE	426 μA
Rx Data Path	RX (LISTEN)	19.7 mA
Tx Data Path	TX (+0 dBm)	17.4 mA
	TX (-1 dBm)	16.5 mA
	TX (-3 dBm)	15.2 mA
	TX (-5 dBm)	13.9 mA
	TX (-7 dBm)	12.5 mA
	TX (-10 dBm)	11.2 mA
	TX (-15 dBm)	9.9 mA
	TX (-25 dBm)	8.5 mA
Flash	POWER_DOWN	9 μA
	STANDBY	25 μA
	READ	7 mA
	WRITE	12 mA
	ERASE	12 mA
LED0 (Red)	ON	4.3 mA
LED1 (Green)	ON	3.7 mA
LED2 (Blue)	ON	1.7 mA

Table 1: The platform energy sinks, their power states, and the nominal current draws in those states at a supply voltage of 3 V and clock speed of 1 MHz, compiled from the datasheets. †*Assumed.*

2.4 Exposing and Tracking Power States

Tracking power states involves a collaborative effort between device drivers and the OS: we modify the device driver that abstracts a hardware resource to expose the device power states through a simple interface, while the OS tracks and logs the power states across the system.

Quanto defines the `PowerState` interface, shown in Figure 1, and provides a generic component that implements it. A device driver merely declares that it uses this interface and signals hardware power state changes

```
interface PowerState {
  // Sets the powerstate to value.
  async command void set(powerstate_t value);

  // Sets the bits represented by mask to value.
  async command void setBits(powerstate_t mask,
    uint8_t offset, powerstate_t value);
}
```

Figure 1: Device drivers must be modified to expose device power states using the `PowerState` interface.

```
async command void Leds.led0On() {
  call Led0PowerState.set(1);
  // Setting pin to low turns Led on
  call Led0.clr();
}

async command void Leds.led0Off() {
  call Led0PowerState.set(0);
  // Setting pin to high turns Led off
  call Led0.set();
}
```

Figure 2: Implementing power state tracking is simple for many devices, like LEDs, and requires signaling power state changes using the `PowerState` interface.

through its simple calls. This approach eliminates state tracking in many device drivers and simplifies the instrumentation of more complex device drivers. Multiple calls to the `PowerState` interface signaling the same state are idempotent: such calls do not result in multiple notifications to the OS.

Figure 2 illustrates the changes to the LED device driver to expose power states. This requires intercepting calls to turn the LED on and off and notifying the OS of these events. For a simple device like the LED which only has two states and whose power states are under complete control of the processor, exposing the power state is a simple and relatively low-overhead matter.

More involved changes to the device driver are needed if a device's power state can change outside of direct CPU control. Flash memory accesses, for example, go through a handshaking process during which the power states and transitions are visible to the processor but not directly controlled by it. Prior to a write request, a flash chip may be in an idle or sleep state. When the processor asserts the flash chip enable line, the flash transitions to a ready state and asserts the ready line. Upon detecting this condition, the processor can issue a write command over a serial bus, framed by a write line assertion. The flash may then signal that it is busy writing the requested data by asserting the busy signal. When finished with the write, the flash asserts the ready signal. In this example, the device driver should monitor hardware handshake lines or use timeouts to shadow and expose the hardware power state.

The glue between the device drivers and OS is a component that exposes the `PowerState` interface to device drivers and provides the `PowerStateTrack` interface, shown in Figure 3, to the OS and application. This component tracks the power states change events and only notifies the OS and registered application listeners when an actual state change occurs. Each time a power state changes, Quanto logs the current value of the energy meter, the time, and the vector of power states.

```
interface PowerStateTrack {
  // Called if an energy sink power state changes
  async event void changed(powerstate_t value);
}
```

Figure 3: The `PowerStateTrack` interface is used by the OS and applications to receive power state change events in real-time.

2.5 Estimating Energy Breakdown

The purpose of estimating the energy breakdown is to attribute to each energy sink its share of the energy consumption. Quanto uses weighted multivariate least squares to estimate the power draw of each energy sink. The input to this offline regression process is a log that records, for each interval during which the power states are same, the aggregate energy consumed during that interval (ΔE), the length of the interval (Δt), and the power states of all devices during the interval ($\alpha_1, \ldots, \alpha_n$).

We estimate the power draw of the i-th energy sink as follows. First we group all intervals from the log that have the same power state j (a particular setting of $\alpha_1, \ldots, \alpha_n$), adding the time t_j and energy E_j spent at that power state. For each power state j, possibly ranging from 1 to 2^n, we determine the average aggregate power y_j:

$$y_j = E_j / t_j,$$

and collect them in a column vector $\mathbf{Y}$ over all j:

$$\mathbf{Y} = \begin{bmatrix} y_1 & \ldots & y_j & \ldots & y_m \end{bmatrix}^T.$$

Due to quantization effects in both our time and energy measurements, the confidence in y_j increases with both E_j and t_j. Correspondingly, we use a weight, $w_j = \sqrt{E_j t_j}$ for each estimate in the regression, and group them in a diagonal weight matrix $\mathbf{W}$. We use the square root because, for a constant power level, E_j and t_j are linearly dependent.

We first collect the observed power states $\alpha_{j,i}$ in a matrix $\mathbf{X}$:

$$\mathbf{X} = \begin{bmatrix} \alpha_{1,1} & \ldots & \alpha_{1,n} \\ \vdots & \ddots & \vdots \\ \alpha_{m,1} & \ldots & \alpha_{m,n} \end{bmatrix}$$

Then, the unknown power draws are estimated:

$$\mathbf{\Pi} = (\mathbf{X}^T\mathbf{W}\mathbf{X})^{-1}\mathbf{X}^T\mathbf{W}\mathbf{Y},$$

and finally, the residual errors are given by:

$$\epsilon = \mathbf{Y} - \mathbf{X}\mathbf{\Pi}$$

3 Activity Tracking

Tracking power states and energy consumption of energy sinks over time shows where and when the energy is going, but leaves a semantic gap to the programmer of *why* the energy is being spent.

The key here is to attribute energy usage to entities – or resource principals – that are meaningful to the programmer. In traditional operating systems, processes or threads combine the roles of protection domain, schedulable unit, and resource principal, but there are many situations in which it is desirable that these notions be independent. This idea was previously explored in the context of high-performance network servers [2] but it is also especially true in networked embedded systems.

We borrow from earlier work the concept of an *activity* as our resource principal. In the Rialto system in particular [19], an activity was defined as the "*the abstraction to which resources are allocated and to which resource usage is charged*." In other words, an activity is a set of operations whose resource consumption should be grouped together. In the environments we consider, where most of the resource consumption does not happen at the CPU, and sometimes not even on the same node that initiated an activity, it is fundamental to support activities that span different hardware components and multiple nodes.

We close the gap of why energy is spent by assigning the energy consumption to activities that are defined by the programmer at a high level. To do this we follow all operations related to an activity across hardware components on a single node and across the network.

3.1 Overview

To account for the resource consumption of activities, we track when a hardware component, or device, is performing operations on behalf of an activity. A useful analogy is to think of an activity as a color, and devices as being painted with the activity's color when working on its behalf. By properly recording devices' successive colors over time and their respective resource consumptions, we can assign to each activity its share of the energy usage.

Figure 4 shows an example of how activities can span multiple devices and nodes. In the figure, the programmer marks the start of an activity by assigning to the CPU the *sensing* activity ("painting the CPU red"). We represent activities by activity labels, which Quanto carries

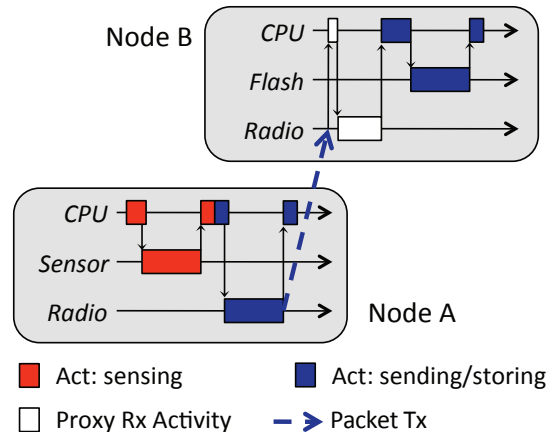


Figure 4: Activity tracking for a sensing, sending, and storing a sample across two nodes. The developer chose sending as a separate activity. Receiving is part of a proxy activity until the CPU can decode the true activity and correctly bind the resource usage.

automatically to causally related operations. For example, when a CPU that is "painted red" invokes an operation on the sensor, the CPU paints the sensor red as well. The programmer may decide to change the CPU activity if it starts work on behalf of a new logical activity, such as when transitioning from sensing to sending (red to blue in the figure). Again the system will propagate the new activity to other devices automatically.

This propagation includes carrying activity labels on network messages, such that operations on node B can be assigned to the activity started on node A. This example also highlights an important aspect of the propagation, namely *proxy* activities. When the CPU on node B receives an interrupt indicating that the radio is starting to receive a packet, the activity to which the receiving belongs is not known. This is generally true in the case of interrupts and external events. Proxy activities are a solution to this problem. The resources used by a proxy activity are accounted for separately, and then assigned to the real activity as soon as the system can determine what this activity is. In this example the CPU can determine that it should be colored blue as soon as it decodes the activity label in the radio packet. It terminates the proxy activity by *binding* it to the blue activity.

The programmer can define the granularity of activities in a flexible way, guided by how she wants to divide the resource consumption of the system. Some operations do not clearly belong to specific activities, such as data structure maintenance or garbage collection. One option is to give these operations their own activities, representing this fact explicitly.

The mechanisms for tracking activities are divided into three parts, which we describe in more detail next,

```
interface SingleActivityDevice {
  // Returns the current activity
  async command act_t get();

  // Sets the current activity
  async command void set(act_t newActivity);

  // Sets the current activity and indicates
  // that the previous activity's resource
  // usage should be charged to the new one
  async command void bind(act_t newActivity);
}
```

Figure 5: The `SingleActivityDevice` interface. This interface represents hardware components that can only be part of one activity at a time, such as the CPU or the transmit part of the radio.

```
interface MultiActivityDevice {
  // Adds an activity to the set of current
  // activities for this device
  async command error_t add(act_t activity);

  // Removes an activity from the set of current
  // activities for this device
  async command error_t remove(act_t activity);
}
```

Figure 6: The `MultiActivityDevice` interface. This interface represents hardware components that can be working simultaneously on behalf of multiple activities. Examples include hardware timers and the receiver circuitry in the radio (when listening).

in the context of TinyOS: (i) an API that allows the programmer to create meaningful activity labels, (ii) a set of mechanisms to propagate these labels along with the operations that comprise the activity, (iii) and a mechanism to account for the resources used by the activities.

3.2 API

We represent activity labels with pairs of the form ⟨*origin node*:*id*⟩, where *id* is a statically defined integer, and *origin node* indicates the node where the activity starts.

We provide an API that allows the assignment of activity labels to devices over time. This API is shown in Figures 5 and 6, respectively, for devices that can only be performing operations on behalf of one, or possibly multiple activities simultaneously. Most devices, including CPUs, are `SingleActivityDevices`.

There are two classes of users for the API, application programmers and system programmers. Application programmers simply have to define the start of high-level activities, and assign labels to the CPU immediately before their start. System programmers, in turn, use the API to propagate activities in the lower levels of the system such as device drivers. We instrumented core parts of the OS, such as interrupt routines, the scheduler, arbiters [21], the network stack, radio, and the

```
task void sensorTask() {
  call CPUActivity.set(ACT_HUM);
  call Humidity.read();
  call CPUActivity.set(ACT_TEMP);
  call Temperature.read();
}

void sendIfDone() {
  if (sensingDone) {
    call CPUActivity.set(ACT_PKT);
    post sendTask();
    sensingDone = 0;
  }
}
```

Figure 7: Excerpt from a sense-and-send application, showing how an application programmer "paints" the CPU to start tracking activities.

timer system. Figure 7 shows an excerpt of a sense-and-send application similar to the one described in [21], in which the application programmer "paints" the CPU using the `CPUActivity.set` method (an instance of the `SingleActivityDevice` interface) before the start of each logical activity. The OS takes care of correctly propagating the labels with the following execution.

3.3 Propagation

Once we have application level activities set by the application programmer, the OS has to carry activity labels to all operations related to each activity. This involves 4 major components: (i) transfer activity labels across devices, (ii) transfer activity labels across nodes, (iii) bind proxy activities to real activities when interrupts occur, and (iv) follow logical threads of computation across several control flow deferral and multiplexing mechanisms.

To transfer activity labels across devices our instrumentation of TinyOS uses the `Single-` and `MultiActivityDevice` APIs. Each hardware component is represented by one instantiation of such interfaces, and keeps the activity state for that component globally accessible to code. The CPU is represented by a `SingleActivityDevice`, and is responsible for transferring activity labels to and from other devices. An example of this transfer is shown in Figure 8, where the code "paints" the radio device with the current CPU activity. Device drivers must be instrumented to correctly transfer activities between the CPU and the devices they manage. In our prototype implementation we instrumented several devices, including the CC2420 radio and the SHT11 sensor chip. Also, we instrumented the Arbiter abstraction [21], which controls access to a number of shared hardware components, to automatically transfer activity labels to and from the managed device.

To transfer activity labels across nodes, we added a hidden field to the TinyOS Active Message (AM) implementation (the default communication abstraction).

When a packet is submitted to the OS for transmission, the packet's activity field is set to the CPU's current activity. This ensures the packet is colored the same as the activity which initiated its submission. We currently encode the labels as 16-bit integers representing both the node id and the activity id, which is sufficient for networks of up to 256 nodes with 256 distinct activity ids. Upon decoding a packet, the AM layer on the receiving node sets the CPU activity to the activity in the packet, and binds resources used between the interrupt for the packet reception and the decoding to the same activity.

More generally, this type of resource binding is done when we have interrupts. Our prototype implementation uses the Texas Instruments MSP430F1611 microcontroller. Since TinyOS does not have reentrant interrupts on this platform, we statically assign to each interrupt handling routine a fixed proxy activity. An interrupt routine temporarily sets the CPU activity to its own proxy activity, and the nature of interrupt processing is such that very quickly, in most cases, we can determine to which real activity the proxy activity should be bound. One example is the decoding of the radio packets at the Active Message layer. Another example is an interrupt caused by a device signaling the completion of a task. In this case, the device driver will have stored locally both the state required to process the interrupt and the activity to which this processing should be assigned.

Lastly, the propagation of activity labels should follow the control flow of the logical threads of execution across deferral and multiplexing mechanisms. The most important and general of these mechanisms in TinyOS are *tasks* and *timers*.

TinyOS has a single stack, and uses an event-based execution model to multiplex several parallel activities among its components. The schedulable unit is a *task*. Tasks run to completion and do not preempt other tasks, but can be preempted by asynchronous events triggered by interrupts. To achieve high degrees of concurrency, tasks are generally short lived, and break larger computations in units that schedule each other by posting new tasks. We instrumented the TinyOS scheduler to save the current CPU activity when a task is posted, and restore it just before giving control to the task when it executes, thereby maintaining the activities bound to tasks in face of arbitrary multiplexing. Timers are also an important control flow deferral mechanism, and we instrumented the virtual timer subsystem to automatically save and restore the CPU activity of scheduled timers.

There are other less general structures that effectively defer processing of an activity, such as forwarding queues in protocols, and we have to instrument these to also store and restore the CPU activity associated with the queue entry. As we show in Section 4, changes to support propagation in a number of core OS services were small and localized.

```
void loadTXFIFO() {
  ...
  //prepare packet
  ...
  call RadioActivity.set(call CPUActivity.get());
  call TXFIFO.write((uint8_t*)header,
                    header->length - 1);
}
```

Figure 8: Excerpt from the CC2420 transmit code that loads the TXFIFO with the packet data. The instrumentation sets the `RadioActivity` to the current value of the `CPUActivity`.

```
interface SingleActivityTrack {
  async event void changed(act_t newActivity);
  async event void bound(act_t newActivity);
}
interface MultiActivityTrack {
  async event void added(act_t activity);
  async event void removed(act_t activity);
}
```

Figure 9: `Single-` and `MultiActivityTrack` interfaces provided by device abstractions. Different accounting modules can listen to these events.

3.4 Recording and Accounting

The final element of activity tracking is recording the usage of resources for accounting and charging purposes. Similarly to how we track power states, we implement the observer pattern through the `Single-ActivityTrack` and `MultiActivityTrack` interfaces (Figure 9). These are provided by a module that listens to the activity changes of devices and is currently connected to a logger. In our prototype we log these events to RAM and do the accounting offline. For single-activity devices, this is straightforward, as time is partitioned among activities. For multi-activity devices, the the log records the set of activities for a device over time, and how to divide the resource consumption among the activities for each period is a policy decision. We currently divide resources equally, but other policies are certainly possible.

4 Evaluation

In this section we first look at two simple applications, *Blink* and *Bounce*, that illustrate how Quanto combines activity tracking, power-state tracking, and energy metering into a complete energy map of the application. We use the first, *Blink*, to calibrate Quanto against ground truth provided by an oscilloscope, and as an example of a multi-activity, single-node application. We use the second, *Bounce*, as an example with activities that span dif-

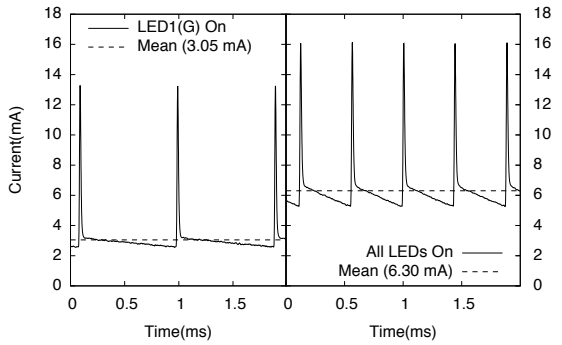


Figure 10: Current over time for two states of Blink recorded with the oscilloscope, showing the mean current and the iCount pulses that Quanto accumulates.

ferent nodes. We then look at three case studies in which Quanto exposes real-world effects and costs of application design decisions, and lastly we quantify some of the costs involved in using Quanto itself. In these experiments processed Quanto data with a set of tools we wrote to parse and visualize the logs. We used GNU Octave to perform the regressions.

4.1 Calibration

We set up a simple experiment to calibrate Quanto against the ground truth provided by a digital oscilloscope. The goal is to establish that Quanto can indeed measure the aggregate energy used by the mote, and that the regression does separate this energy use by hardware components.

We use Blink, the *hello world* application in TinyOS. Blink is very simple; it starts three independent timers with intervals of 1, 2, and 4s. When these timers fire, the red, green, and blue LEDs are toggled, such that in 8 seconds Blink goes through 8 steady states, with all combinations of the three LEDs on and off. The CPU is in its sleep state during these steady states, and only goes active to perform the transitions.

Using the Hydrowatch board (*cf.* Section 2.2), we connected a Tektronix MSO4104 oscilloscope to measure the voltage across a 10Ω resistor inserted between iCount circuit and the mote power input. We measured the voltage provided by the regulator for the mote to be 3.0V.

We confirmed the result from [9] that the switching frequency of iCount varies linearly with the current. Figure 10 shows the current for two sample states of Blink. This curve has a wealth of information: from it we can derive both the switching frequency of the regulator, which is what Quanto measures directly, and the actual average current, I_{avg}. We verified over the 8 power states that I_{avg}, in mA, and the switching frequency f_{iC}, in kHz, have a linear dependency given by $I_{avg} = 2.77 f_{iC} - 0.05$, with an R^2 value of 0.99995. We can infer from this that each iCount pulse corresponds, in this hardware, at 3 V, to 8.33 μJ. We also verified that I_{avg} was stable during each interval.

$\mathbf{X}$				$\mathbf{Y}$		$\mathbf{X\Pi}$
L0	L1	L2	C	$I(mA)$		$I(mA)$
0	0	0	1	0.74		0.79
1	0	0	1	3.32		3.29
0	1	0	1	3.05		3.02
1	1	0	1	5.53		5.53
0	0	1	1	1.62		1.62
1	0	1	1	4.15		4.12
0	1	1	1	3.88		3.85
1	1	1	1	6.30		6.36

	$\mathbf{\Pi}$
	$I(mA)$
LED0	2.50
LED1	2.23
LED2	0.83
Const.	0.79

Table 2: Oscilloscope measurements of the current for the steady states of Blink, and the results of the regression with the current draw per hardware component. The relative error ($\|\mathbf{Y} - \mathbf{X\Pi}\|/\|\mathbf{Y}\|$) is 0.83%.

Lastly we tested the regression methodology from Section 2.5, using the average current measured by the oscilloscope in each state of Blink and the external state of the LEDs as the inputs. We also added a constant term to account for any residual current not captured by the LED state. Table 2 shows the results, and the small relative error indicates that for this case the linearity assumptions hold reasonably well, and that the regression is able to produce a good breakdown of the power draws per hardware device.

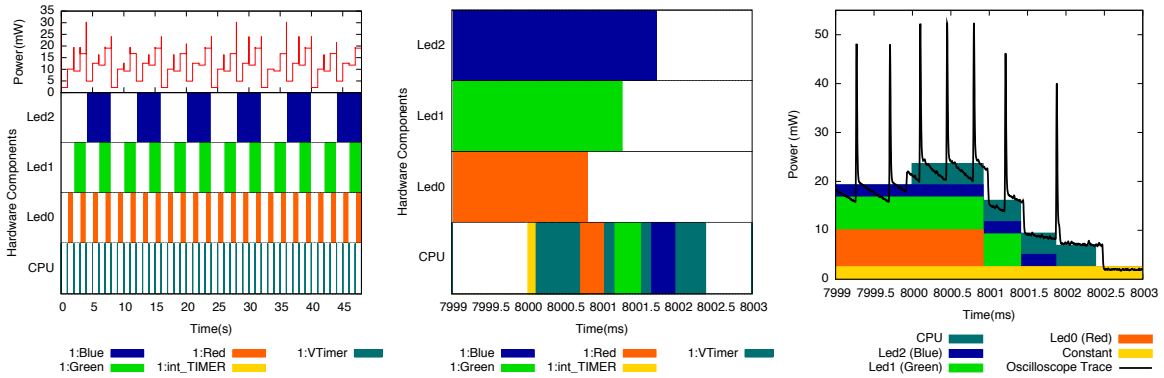
4.2 Two Illustrative Examples

4.2.1 Blink

We instrumented Blink with Quanto to verify the results from the calibration and to demonstrate a simple case of tracking multiple activities on a single node. We divided the application into 3 main activities: Red, Green, and Blue, which perform the operations related to toggling each LED. Each LED, when on, gets labeled with the respective activity by the CPU, such that its energy consumption can be charged to the correct activity. We also created an activity to represent the managing of the timers by the CPU (VTimer). We recorded the power states of each LED (simply on and off), and consider the CPU to only have two states as well: active, and idle.

Figures 11(a) and (b) show details of a 48-second run of Blink. In these plots, the X axis represents time, and each color represents one activity. The lower part of (a) shows how each hardware component divided its time among the activities. The topmost portion of the graph shows the aggregate power draw measured by iCount. There are eight distinct stable draws, corresponding to the eight states of the LEDs.

Part (b) zooms in on a particular state transition spanning 4 ms, around 8 s into the trace, when all three LEDs



(a) Power draw measured by Quanto and the activities over time for each hardware component.

(b) Detail of a transition from all on to all off, showing the activities for each hardware component.

(c) Stacked power draw for the hardware components, with values from the regression, overlaid with the oscilloscope-measured power.

Figure 11: Activity and power profiles for a 48-second run of the Blink application on the Hydrowatch platform.

Activities	Hardware Components - Time(s) LED0	LED1	LED2	CPU
1:Red	24.01	0	0	0.0176
1:Green	0	24.00	0	0.0091
1:Blue	0	0	24.00	0.0045
1:Vtimer	0	0	0	0.0450
1:int_Timer	0	0	0	0.0092
1:Idle	23.99	24.00	24.00	47.9169
Total	48.00	48.00	48.00	48.0024

(a) Time break down.

	Hardware Components - $\mathbf{\Pi}$ LED0	LED1	LED2	CPU	*Const.*
I_{avg}(mA)	2.51	2.24	0.83	1.43	0.83
P_{avg}(mW)	7.53	6.71	2.49	4.29	2.48

(b) Result of the regression.

	$\sum E_{HW}$(mJ)
LED0	180.71
LED1	161.06
LED2	59.84
CPU	0.37
Const.	119.26
Total	521.23

(c) Total Energy per Hardware Component.

Activities	$\sum E_{act}$(mJ)
1:Red	180.78
1:Green	161.10
1:Blue	59.86
1:Vtimer	0.19
1:int_Timer	0.04
1:Idle	0.00
Const.	119.26
Total	521.23

(d) Total Energy per Activity.

Table 3: Where the joules have gone in *Blink*. The tables show how activities spend time on hardware components (a), the regression results (b), and a break down of the energy usage by activity (c) and hardware component (d).

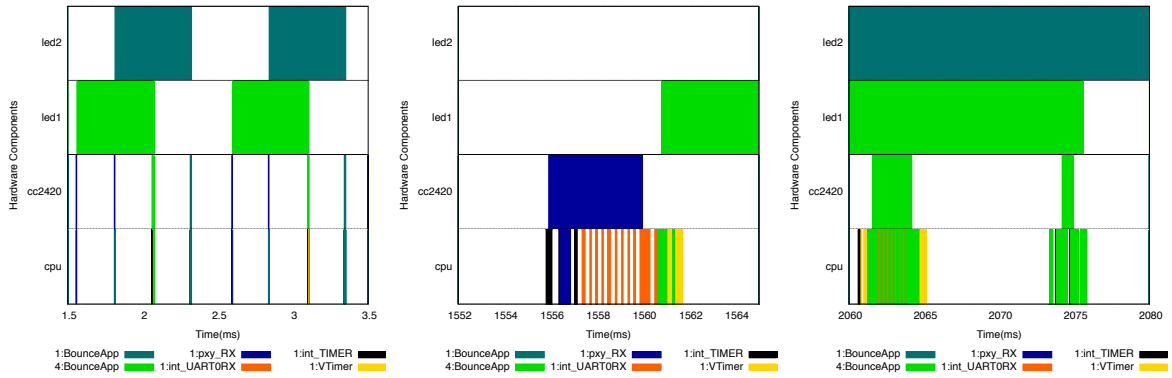
simultaneously go from their on to off state, and cease spending energy on behalf of their respective activities. At this time scale it is interesting to observe the CPU activities. For clarity, we did not aggregate the proxy activities from the interrupts into the activities they are bound to. At 8.000 s the timer interrupt fires, and the CPU gets labeled with the int_TIMERB0 and VTimer activities. VTimer, after examining the scheduled timers, yields to the Red, Green, and Blue activities in succession. Each activity turns off its respective LED, clears its activities, and sets its power state to off. VTimer performs some bookkeeping and then the CPU sleeps.

Table 3(a) shows, for the same run, the total time when each hardware component spent energy on behalf of each activity. The CPU is active for only 0.178% of the time. Also, although the LEDs stay on for the same amount of time, they change state a different number of times, and the CPU time dedicated to each corresponding activity reflects that overhead.

We ran the regression as described in Section 2.5 to identify the power draw of each hardware component. Table 3(b) shows the result in current and power. This information, combined with the time breakdown, allows us to compute the energy breakdown by hardware component (c), and by activity (d). The correlation between the corresponding components of Table 3(b) and the current breakdown in Table 2 is 0.99988. Note that we don't have the CPU component in the oscilloscope measurements because it was hard to identify in the oscilloscope trace exactly when the CPU was active, something that is easy with Quanto.

From the power draw of the individual hardware components we can reconstruct the power draw of each power state and verify the quality of the regression. The relative error between the total energy measured by Quanto and the energy derived from the reconstructed power state traces was 0.004% for this run of Blink.

Figure 11(c) shows a stacked breakdown of the measured energy envelope, reconstructed from the power state time series and the results of the regression. The shades in this graph represent the different hardware components, and at each interval the stack shows which components are active, and in what proportion they con-



(a) A 2-second window on a run of Bounce at a node with id 1.

(b) Detail of a packet reception with an activity label from node 4.

(c) Detail of a packet transmission on node 1 as part of the activity started at node 4.

Figure 12: Activity tracking on Bounce. Each packet carries the activity current at the time it was generated, and the receiving node executes some operations as part of that remote activity.

tribute to the overall energy consumption. The graph also shows an overlaid power curve measured with the oscilloscope for the same run. The graph shows a very good match between the two sources, both in the time and energy dimensions. We can notice small time delays between the two curves, on the order of 100 μs, due to the time Quanto takes to record a measurement.

4.2.2 Bounce

The second example we look at illustrates how Quanto keeps track of activities across nodes. Bounce is a simple application in which two nodes keep exchanging two packets, each one originating from one of the nodes. In this example we had nodes with ids 1 and 4 participate. All of the work done by node 1 to receive, process, and send node 4's original packet is attributed to the '4:BounceApp' activity. Although this is a trivial example, the same idea applies to other scenarios, like protocol beacon messages and multihop routing of packets.

Figure 12 shows a 2-second trace from node 1 of a run of Bounce. The log at the other node is symmetrical. On part (a) we see the entire window, and the activities by the CPU, the radio, and two LEDs that are on when the node has "possession" of each packet. In this figure, node 1 receives a packet which carries the 4:BounceApp activity, and turns LED1 on because of that. The energy spent by this LED will be attributed to node 4's original activity. The node then receives another packet, which carries its own 1:BounceApp activity. LED2's energy spending will be assigned to node 1's activity, as well as the subsequent transmission of this same packet.

Figures 12(b) and (c) show in detail a packet reception and transmission, and how activity tracking takes place in these two operations. Again, we keep the interrupt proxy activities separated, although when accounting for resource consumption we should assign the consumption of a proxy activity to the activity to which it binds. The receive operation starts with a timer interrupt for the start of frame delimiter, followed by a long transfer from the radio FIFO buffer to the processor, via the SPI bus. This transfer uses an interrupt for every 2 bytes. When finished, the packet is decoded by the radio stack, and the activity in the packet can be read and assigned to the CPU. The CPU then "paints" the LED with this activity and schedules a timer to send the packet.

Transmission in Bounce is triggered by a timer interrupt that was scheduled upon receive. The timer carries and restores the activity, and "paints" the radio. There are two main phases for transmission. First, the data is transferred to the radio via the SPI bus, and then, after a backoff interval, the actual transmission happens. When the transmission is done, the CPU then turns the LED off and sets its activity to idle.

4.3 Case Studies

Quanto allows a developer to precisely understand and quantify the effects of design decisions, and we discuss three case studies from the TinyOS codebase.

The first one is an investigation of the effect of interference from an 802.11 b/g network on the operation of low-power listening [25]. Low-power listening (LPL) is a family of duty-cycle regimes for the radio in which the receiver stays mostly off, and periodically wakes up to detect whether there is activity on the channel. If there is, it stays on to receive packets, otherwise it goes back to sleep. In the simplest version, a sender must transmit a packet for an interval as long as the receiver's sleep interval. A higher level of energy in the channel, due to interference from other sources, can cause the receiver to falsely detect activity, and stay on unnecessarily. Since

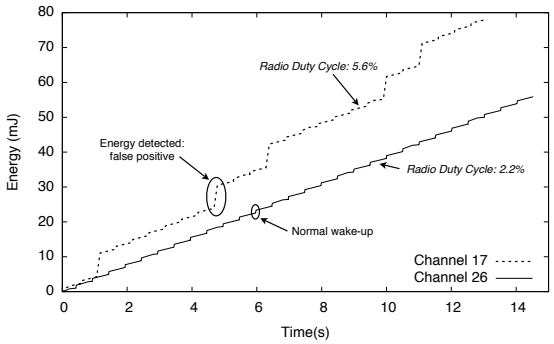


Figure 13: 802.11 b/g interference on the mote 802.15.4 radio. In the top curve the mote was set to the 802.15.4 channel 17, and in the bottom curve, to channel 26. These are, respectively, the closest to and furthest from the 802.11 b channel 6 which was used in the experiment.

802.11 b/g and 802.15.4 radios share the 2.4 GHz band, and the former generally has much higher power than the latter, this scenario can be quite common. We used Quanto to measure the impact of such interference. We set an 802.11 b access point to operate on channel 6, with central frequency of 2.437 GHz, and programmed a TinyOS node to listen on LPL mode, first on the 802.15.4 channel 17 (central frequency 2.453 GHz), and then on channel 26 (central frequency 2.480 GHz). We set the TinyOS node to sample the channel every 500 ms, and placed it 10 cm away from the access point. We collected data for 5 14-second periods at each of the two channels.

We verified a significant impact of the interference: when on channel 17, the node falsely detected activity on the channel 17.8% of the time, had a radio duty cycle of 5.58±0.005%, and an average power draw of 1.43±0.08 mW. The nodes on channel 26, on the other hand, detected *no* false positives, had a duty cycle of 2.22±0.0027%, and an average power draw of 0.919±0.006 mW.

Figure 13 shows one measurement at each channel. The steps on the channel 17 curve are false positives, and have a marked effect on the cumulative energy consumption. Using Quanto, we estimated the current for the radio listen mode to be 18.46 mA, with a power draw of 61.8 mW (this particular mote was operating with a 3.35V switching regulator). Figure 14 shows two sampling events on channel 17. For both the radio and the CPU, the graph shows the power draw when active, and the respective activities. We can see the VTimer activity, which schedules the wake-ups, and the proxy receive activity, which doesn't get bound to any subsequent higher level activity. This is a simple example, but Quanto would be able to distinguish these activities even if the node were performing other tasks.

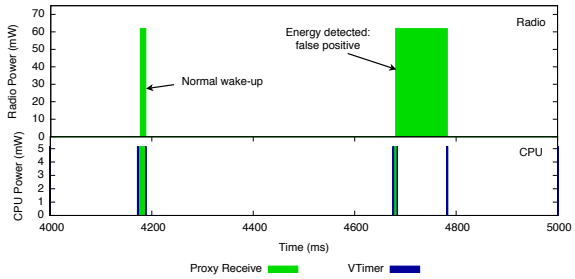


Figure 14: Detail of a normal wake-up period with no activity, in which the radio wakes up and returns to sleep, and of a false-positive activity detection. In the latter, the CPU keeps the radio on for about 100 ms, and turns it off when the timer expires and no packet was received.

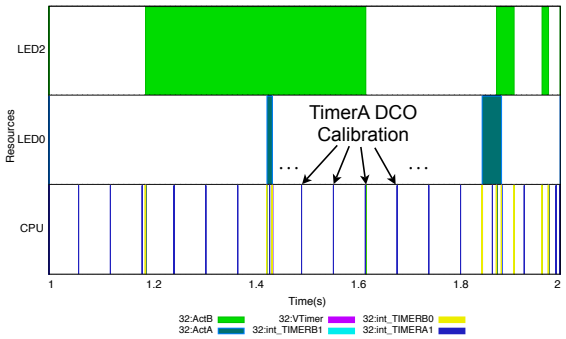


Figure 15: An unexpected result from instrumenting a simple application with Quanto: we noticed that a particular timer interrupt was firing 16 times per second for oscillator calibration, even when such calibration was unnecessary.

Our second example concerns an effect we noticed when we instrumented a simple timer-based application on a single node. A particular timer interrupt, TimerA1, was firing repeatedly at 16Hz, as can be seen in Figure 15. This timer is used for calibrating a digital oscillator, which is not needed unless the node requires asynchronous serial communication. However, it was set to be always on, a behavior that surprised many of the TinyOS developers. The lack of visibility into the system made this behavior go unnoticed.

Our last example studies the effect of a particular setting of the radio stack: whether the CPU communicates with the radio chip using interrupts or a DMA channel. Figure 16 shows the timings captured by Quanto for a packet transmission, using both settings.

From the figure it is apparent that the DMA transfer is at least twice as fast as the interrupt-driven transfer. This has implications on how fast one can send packets, but more importantly, can influence the behavior of the

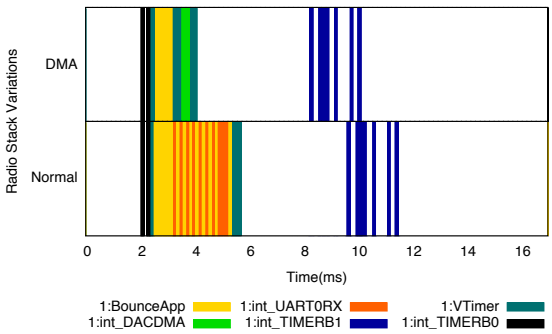


Figure 16: Timing behavior of a packet transmission using interrupt-driven and DMA-based communication between the CPU and the radio chip. Quanto allows the developer to understand the precise timing behavior of both options.

MAC protocol. If two nodes A and B receive the same packet from a third node, and need to respond to it immediately, and if A uses DMA while B uses the interrupt-driven communication, A will gain access to the medium more often than B, subverting MAC fairness.

4.4 Costs

We now look at some of the costs associated with our prototype implementation of Quanto. These are summarized in Table 4.

Cost of logging. The design of Quanto decouples generating event information, like activity and power state changes, from tracking the events. We currently record a log of the events for offline processing. The cost of logging is divided in two parts, one synchronous and one asynchronous. Recording the time and energy for each event has to be done synchronously, as close to the event as possible. Dealing with the recorded information can be done asynchronously.

It is very important to minimize the cost of synchronously recording each sample, as this both limits the rate at which we can capture successive events, and delays operations which must be processed quickly. Our current implementation records a 12-byte log entry for each event, described in Figure 17. We measured the cost of logging to RAM to be 101.7 μs, using the same technique as in [9]. At 1MHz, this translates to 102 cycles. This time includes 24 μs to read the iCount value, and 19 μs to read a timer value.

Because Quanto uses the CPU to keep track of state and to log changes to state, using it incurs a cost by delaying operations on the CPU, and spending more energy. For the run of Blink in Section 4.2, we logged 597

```
typedef struct entry_t {
  uint8_t type;      // type of the entry
  uint8_t res_id;    // hardware resource for entry
  uint32_t time;     // local time of the node
  uint32_t ic;       // icount: cumulative energy
  union {
    uint16_t act;          //for ctx changes
    uint16_t powerstate;   //for powerstate changes
  };
} entry_t;
```

Figure 17: The structure for the activity and powerstate log entry.

Buffer Size	800 samples
Sample Size	12 bytes
Cost of Logging	102 cycles @ 1MHz
Call Overhead	41 cycles
Read Timer	19 cycles
Read iCount	24 cycles
Others	18 cycles

Table 4: Costs associated with logging to RAM.

messages over 48 seconds. The total time spent on the logging itself was 60.71 ms, corresponding to 71.05% of the *active* CPU time, but only 0.12% of the total CPU time. The total energy spent with logging, assuming that logging is using the CPU and the *Constant* terms in the regression results, was 0.41 mJ, or 0.08% of the total energy spent. Although the 71% number is high, the majority of applications in these sensor network platforms strive to reduce the CPU duty cycle to save energy, and we expect the same trend of long idle periods to amortize the cost of logging.

The above numbers only concern the synchronous part. We still have to get the data out of the node for the current approach of offline analysis. We have two implementations for this. The first records messages to a fixed buffer in RAM that holds 800 log entries, periodically stops the logging, and dumps the information to the serial port or to the radio. The advantage of this is that the cost of logging, during the period being monitored, is only the cost of the synchronous part.

The second approach allows continuous logging. The processor still collects entries to the memory buffer, and schedules a low priority task to empty the log. This happens only when the CPU would otherwise be idle. Messages are written directly to an output port of the microprocessor, which drives an external synchronous serial interface. Like the Unix `top` application, Quanto can account for its own logging in this mode as its own activity. For the applications we instrumented, it used between 4 and 15% of the CPU time.

The rate of generated data from Quanto largely depends on the nature of the workload of the application. For the classes of applications that are common in em-

	Files	Diff LOC	
Modified Code			
Tasks	2	25	Concurrency
Timers	2	16	Deferral
Arbiter	5	34	Locks
Interrupts	11	88	
Active Msg.	2	8	Link Layer
LEDs	2	33	Device Driver
CC2420 Radio	11	105	Device Driver
SHT11	3	10	Sensor
New code	28	1275	Infrastructure

Table 5: The cost of instrumenting most core primitives for activity and power tracking in TinyOS, as well as some representative device drivers, is low in terms of lines of code. New code represents the infrastructure code for keeping track of and logging activities and power states.

bedded sensor networks, of low data-rate and duty cycle, we believe the overheads are acceptable.

Instrumentation costs. Finally, we look at the burden to instrument a system like TinyOS to allow tracking and propagation of activities and power states. Table 5 lists the main abstractions we had to instrument in TinyOS to achieve propagation of activity labels in our platform, and shows that the changes are highly localized and relatively small in number of lines of code.

The complexity of the instrumentation task varies, and some device drivers with shadowed state that represents volatile state in peripherals can be more challenging to instrument. The CC2420 radio is a good example, as it has several internal power states and does some processing without the CPU intervention. Other devices, like the LEDs and simple sensors, are quite easier. We found that once the system is instrumented, the burden to the application programmer is small, since all that needs to be done is marking the beginning of relevant activities, which will be tracked and logged automatically.

5 Discussion

We now discuss some of the the design tradeoffs and limitations of the approach, and some research directions enabled by this work.

5.1 Design Tradeoffs

Logging vs. counting. Quanto currently logs every power state and activity context change which can result in large volume of trace data. The data are useful for reconstructing a fine-grained timeline and tracing causal connections, but this level of detail may be unnecessary in many cases. The design, however, clearly separates the event generation from the event consumption. An alternative would be to maintain a set counters on the nodes, accumulating time and energy spent per activity. In our initial exploration we decided to examine the full dataset offline, and leave as future work to explore performing the regression and accounting of resources online, which would make the memory overhead fixed and practically eliminate the logging overhead.

Activity model. An important design decision in Quanto is that activities are not hierarchical. While giving more flexibility, representing hierarchies would mean that the system would propagate stacks of activity labels instead of single labels, a significant increase in overhead and complexity. If a module C does work on behalf of two activities, A and B, the instrumenter has two options: to give C its own activity, or to have C's operations assume the activity of the caller.

Platform hardware support. All of the data in this paper were collected using the HydroWatch platform but our experiences suggested that a more tailored design would be useful. In particular, we had the options of storing the logs in RAM, which has little impact but limited space, or logging to a processor port, which has a slightly higher cost and can be intrusive at very high loads. We have designed a new platform tailored for profiling with a fast, 128 KB-deep FIFO for full speed logging with very little overhead, which we plan to use on future experiments.

5.2 Limitations

Constant per-state power draws. The regression techniques used to estimate per-component energy usage assume the power draw of a hardware component is approximately constant in each power state. Fortunately, we verified that this assumption largely holds for the platform we instrumented, by looking at different length sampling intervals for each state. The regression may not work well when this assumption fails, but we leave quantifying this for future work.

Linear independence. The regression techniques also assume that tracking power states over time produces a set of linearly independent equations. If this is not the case, for example if unrelated actions always occur together, then regression is unlikely to disambiguate their energy usage. As a work around, custom routines can be written to exercise different power states independently.

Modifications to systems. Quanto requires the OS, including device drivers, and applications, to be modified to perform tracking. The modifications to the system, however, can be shared among all applications, and the modifications to applications are, in most cases, simple. Device drivers have to be modified so that they expose

the power states of the underlying hardware components. If hardware power states are not observable, estimation errors may occur.

Energy usage visibility. Our approach may not generalize to systems with sophisticated power supply filtering (e.g. power factor correction or large capacitors) because these elements introduce a potentially non-linear phase delay between real and observed energy usage over short time scales, making it difficult to correlate short-lived activities with their energy usage.

Hardware energy metering. Our proposed approach requires hardware support for energy metering, which may not be available on some platforms. Fortunately, the energy meter design we use may be feasible on many systems that use pulse-frequency modulated switching regulators. However, even if hardware-based energy metering is not available, a software-based approach using hardware power models may still provide adequate visibility for some applications.

5.3 Enabled Research

Finding energy leaks. A situation familiar to many developers is discovering that an application draws too much power but not knowing why. Using Quanto, developers can visualize energy usage over time by hardware component, allowing one to work backward to find the offending code that caused the energy leak.

Tracking butterfly effects. In many distributed applications, an action at one node can have network-wide effects. For example, advertising a new version of a code image or initiating a flood will cause significant network-wide action and energy usage. Even minor local actions, like a routing update, can ripple through the entire network. Quanto can trace the causal chain from small, local cause to large, network-wide effect.

Real time tracking. An extension of the framework can include performing the regression online, and replacing the logging with accumulators for time and energy usage per activity. This approach would have significantly reduced bandwidth and storage requirements, and could be used as an always on, network-wide energy profiler analogous to `top`.

Enery-Aware Scheduling. Since Quanto already tracks energy usage by activity, an extension to the operating system scheduler would enable energy-aware policies like equal-energy scheduling for threads, rather than equal-time scheduling.

Continuous Profiling. Quanto log entries are lightweight enough that continuous profiling is possible with even a modest speed logging back-channel [1].

6 Related Work

Our techniques borrow heavily from the literature on energy-aware operating system, power simulation tools, power/energy metering, power profiling, resource containers, and distributed tracing.

ECOSystem [37] proposes the Currentcy model which treats energy as a first class resource that cuts across all existing system resources, such as CPU, disk, memory, and the network in a unified manner. Quanto leverages many of the ideas developed in ECOSystem, like tracking power states to allocate energy usage or employing resource containers as the principal to which resource usage is charged. But there are important differences as well. ECOSystem uses offline profiling to relate power state and power draw, and uses a model for runtime operation. In contrast, Quanto tracks the actual energy used at runtime, which is useful when environmental factors can affect energy availability and usage. While ECOSystem tracks energy usage on a single node, Quanto transparently tracks energy usage across the network, which allows network-wide effects to be measured. Finally, the focus of the two efforts is different although similar techniques are used in both systems.

Eon is a programming language and runtime system that allows paths or flows through the program to be annotated with different energy states [29]. Eon's runtime then chooses flows to execute, and their rates of execution, to maximize the quality of service under available energy constraints. Eon, like Quanto, uses real-time energy metering but attributes energy usage only to flows, while Quanto attributes usage to hardware, activity, and time.

Several power simulation tools exist that use empirically-generated models of hardware behavior. PowerTOSSIM [28] uses same-code simulation of TinyOS applications with power state tracking, combined with a power model of the different peripheral states, to create a log of energy usage. PowerTOSSIM provides visibility into the power draw based on its model of the hardware, but it does not capture the variability common in real hardware or operating environments, or simulate a device's interactions with the real world. Quanto also addresses a different problem than PowerTOSSIM: tracing the energy usage of logical activities rather than the time spent in software modules.

The challenge in taking measurements in low-power, embedded systems that exhibit bursty operation is that until recently, the performance of available metering options was simply too poor, and the power cost was simply too high, to use in actual deployments. Traditional instrument-based power measurements are useful for design-time laboratory testing but impractical for everyday run-time use due to the cost of instru-

ments, their physical size, and their poor system integration [14, 12, 35]. Dedicated power metering hardware can enable run-time energy metering but they too come with the expense of increased hardware costs and power draws [5, 18]. Using hardware performance counters as a proxy power meter is possible on high-performance microprocessors like the Intel Pentium Pro [20] and embedded microprocessors like the Intel PXA255 [7]. Quanto addresses these challenges with iCount, a new design based on a switching regulator [9].

Of course, if a system employs only one switching regulator, then the energy usage can be measured only in the aggregate, rather than by hardware component. This aggregated view of energy usage can present some tracking challenges as well. One way to track the distinct power draws of the hardware components is to instrument their individual power supply lines [34, 30]. These approaches, however, are best suited to bench-scale investigations since they require extensive per-system calibration and the latter requires considerable additional hardware which would dominate the system power budget in our applications.

The Rialto operating system [19] introduced *activities* as the abstraction to which resources are allocated and charged. Resource Containers [2] use a similar notion, and acknowledge that there is a mismatch between traditional OS resource principals, namely threads and processes, and independent activities, especially in high performance network servers. Quanto borrows the concept of activities and extends them across all hardware components and across the nodes in a network.

Several previous works have modeled the behavior of distributed systems as a collection of causal paths including Magpie [3], Pinpoint [6], X-Trace [15], and Pip [27]. These systems reconstruct causal paths using some combinations of OS-level tracing, application-level annotation, and statistical inference. Causeway [4] instruments the FreeBSD OS to automatically carry metadata with the execution of threads and across machines. Quanto borrows from these earlier approaches and applies them to the problem of tracking network-wide energy usage in embedded systems, where resource constraints and energy consumption by hardware devices raise a number of different design tradeoffs.

7 Conclusion

The techniques developed and evaluated in this paper – breaking down the aggregate energy usage of a system by hardware component, tracking causally-connected energy usage of programmer-defined activities, and tracking the network-wide energy usage due to node-local actions – collectively provide visibility into when, where, and why energy is consumed both within a single node and across the network. Going forward, we believe this unprecedented visibility into energy usage will enable empirical evaluation of the energy-efficiency claims in the literature, provide ground truth for lightweight approximation techniques like counters, and enable energy-aware operating systems research.

8 Acknowledgments

This material is based upon work supported by the National Science Foundation under grants #0435454 (“NeTS-NR”) and #0454432 (“CNS-CRI”). This work was also supported by a National Science Foundation Graduate Research Fellowship and a Microsoft Research Graduate Fellowship as well as generous gifts from Hewlett-Packard Company, Intel Research, Microsoft Corporation, and Sharp Electronics.

References

[1] ANDERSON, J. M., BERC, L. M., DEAN, J., GHEMAWAT, S., HENZINGER, M. R., LEUNG, S.-T. A., SITES, R. L., VANDEVOORDE, M. T., WALDSPURGER, C. A., AND WEIHL, W. E. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst. 15*, 4 (1997), 357–390.

[2] BANGA, G., MOGUL, J. C., AND DRUSCHEL, P. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)* (February 1999).

[3] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using magpie for request extraction and workload modelling. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation* (Berkeley, CA, USA, 2004), USENIX Association, pp. 18–18.

[4] CHANDA, A., ELMELEEGY, K., COX, A. L., AND ZWAENEPOEL, W. Causeway: operating system support for controlling and analyzing thexecution of distributed programs. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2005), USENIX Association, pp. 18–18.

[5] CHANG, F., FARKAS, K., AND RANGANATHAN, P. Energy-driven statistical profiling: Detecting software hotspots. In *Workshop on Power-Aware Computer Systems* (feb 2002).

[6] CHEN, M. Y., ACCARDI, A., KICIMAN, E., LLOYD, J., PATTERSON, D., FOX, A., AND BREWER, E. Path-based faliure and evolution management. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2004), USENIX Association, pp. 23–23.

[7] CONTRERAS, G., AND MARTONOSI, M. Power prediction for intel xscale processors using performance monitoring unit events. In *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design* (New York, NY, USA, 2005), ACM, pp. 221–226.

[8] DEAN, J., HICKS, J. E., WALDSPURGER, C. A., WEIHL, W. E., AND CHRYSOS, G. Profileme: hardware support for instruction-level profiling on out-of-order processors. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture* (Washington, DC, USA, 1997), IEEE Computer Society, pp. 292–302.

[9] DUTTA, P., FELDMEIER, M., PARADISO, J., AND CULLER, D. Energy metering for free: Augmenting switching regulators for real-time monitoring. In *IPSN'08: International Conference on Information Processing in Sensor Networks* (2008), pp. 283–294.

[10] DUTTA, P., TANEJA, J., JEONG, J., JIANG, X., AND CULLER, D. A building block approach to sensornet systems. In *Proceedings of the Sixth ACM Conference on Embedded Networked Sensor Systems (SenSys'08)* (Nov. 2008).

[11] ELSON, J., GIROD, L., AND ESTRIN, D. Fine-grained network time synchronization using reference broadcasts. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation* (New York, NY, USA, 2002), ACM, pp. 147–163.

[12] FARKAS, K. I., FLINN, J., BACK, G., GRUNWALD, D., AND ANDERSON, J. M. Quantifying the energy consumption of a pocket computer and a java virtual machine. *SIGMETRICS Perform. Eval. Rev. 28*, 1 (2000), 252–263.

[13] FLINN, J., AND SATYANARAYANAN, M. Energy-aware adaptation for mobile applications. In *Symposium on Operating Systems Principles (SOSP'99)* (1999), pp. 48–63.

[14] FLINN, J., AND SATYANARAYANAN, M. Powerscope: A tool for profiling the energy usage of mobile applications. In *WMCSA '99: Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications* (Washington, DC, USA, 1999), IEEE Computer Society, p. 2.

[15] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-trace: A pervasive network tracing framework. In *NSDI'07: Proceedings of the 4th USENIX/ACM Symposium on Networked Systems Design and Implementation* (2007), USENIX.

[16] HEMPSTEAD, M., TRIPATHI, N., MAURO, P., WEI, G.-Y., AND BROOKS, D. An ultra low power system architecture for sensor network applications. In *ISCA'05: 32nd International Symposium on Computer Architecture* (2005).

[17] HILL, J., AND CULLER, D. E. Mica: a wireless platform for deeply embedded networks. *IEEE Micro 22*, 6 (nov/dec 2002), 12–24.

[18] JIANG, X., DUTTA, P., CULLER, D., AND STOICA, I. Micro power meter for energy monitoring of wireless sensor networks at scale. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks* (New York, NY, USA, 2007), ACM Press, pp. 186–195.

[19] JONES, M. B., LEACH, P. J., DRAVES, R. P., AND BARRERA, J. S. Modular real-time resource management in the rialto operating system. In *HOTOS '95: Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)* (Washington, DC, USA, 1995), IEEE Computer Society, p. 12.

[20] JOSEPH, R., AND MARTONOSI, M. Run-time power estimation in high performance microprocessors. In *Proceedings of the International Symposium on Low Power Electronics and Design* (2001), pp. 135–140.

[21] KLUES, K., HANDZISKI, V., LU, C., WOLISZ, A., CULLER, D., GAY, D., AND LEVIS, P. Integrating concurrency control and energy management in device drivers. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (New York, NY, USA, 2007), ACM, pp. 251–264.

[22] MADDEN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND HONG, W. Tag: a tiny aggregation service for ad-hoc sensor networks. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation* (New York, NY, USA, 2002), ACM, pp. 131–146.

[23] MUSĂLOIU-E., R., JIANG, C.-J. M., AND TERZIS, A. Koala: Ultra-Low Power Data Retrieval in Wireless Sensor Networks. In *Proceedings of the 7th International Symposium on Information Processing in Sensor Networks (IPSN)* (2008).

[24] PETROVA, M., RIIHIJARVI, J., MAHONEN, P., AND LABELLA, S. Performance study of ieee 802.15.4 using measurements and simulations. In *Wireless Communications and Networking Conference 2006 (WCNC 2006)* (April 2006), vol. 1, pp. 487–492.

[25] POLASTRE, J., HILL, J., AND CULLER, D. Versatile low power media access for wireless sensor networks. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys)* (November 2004).

[26] POLASTRE, J., HUI, J., LEVIS, P., ZHAO, J., CULLER, D., SHENKER, S., AND STOICA, I. A unifying link abstraction for wireless sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems* (New York, NY, USA, 2005), ACM, pp. 76–89.

[27] REYNOLDS, P., KILLIAN, C., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: detecting the unexpected in distributed systems. In *NSDI'06: Proceedings of the 3rd conference on 3rd Symposium on Networked Systems Design & Implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 9–9.

[28] SHNAYDER, V., HEMPSTEAD, M., RONG CHEN, B., WERNER-ALLEN, G., AND WELSH, M. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys'04)* (2004).

[29] SORBER, J., KOSTADINOV, A., GARBER, M., BRENNAN, M., CORNER, M. D., AND BERGER, E. D. Eon: a language and runtime system for perpetual systems. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems* (2007), pp. 161–174.

[30] STATHOPOULOS, T., MCINTIRE, D., AND KAISER, W. The energy endoscope: Real-time detailed energy accounting for wireless sensor nodes. In *IPSN'08: International Conference on Information Processing in Sensor Networks* (2008), pp. 383–394.

[31] SZEWCZYK, R., POLASTRE, J., MAINWARING, A., AND CULLER, D. Lessons From A Sensor Network Expedition. In *Proceedings of the First European Workshop on Wireless Sensor Networks (EWSN)* (2004).

[32] TALZI, I., HASLER, A., GRUBER, S., AND TSCHUDIN, C. PermaSense: Investigating Permafrost with a WSN in the Swiss Alps. In *Proceedings of the Fourth Workshop on Embedded Networked Sensors (EmNets)* (2007).

[33] TOLLE, G., AND CULLER, D. Design of an Application-Cooperative Management System for Wireless Sensor Networks. In *Proceedings of the Second European Workshop of Wireless Sensor Netw orks (EWSN)* (2005).

[34] VIREDAZ, M. A., AND WALLACH, D. A. Power evaluation of a handheld computer. *IEEE Micro 23*, 1 (2003), 66–74.

[35] WERNER-ALLEN, G., SWIESKOWSKI, P., AND WELSH, M. Motelab: a wireless sensor network testbed. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks* (Piscataway, NJ, USA, 2005), IEEE Press, p. 68.

[36] YE, W., HEIDEMANN, J., AND ESTRIN, D. An energy-efficient mac protocol for wireless sensor networks. In *INFOCOM'02: The 21st Annual Joint Conference of the IEEE Computer and Communications Societies* (June 2002).

[37] ZENG, H., ELLIS, C. S., LEBECK, A. R., AND VAHDAT, A. Ecosystem: Managing energy as a first class operating system resource. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)* (2002), pp. 123–132.

Leveraging Legacy Code to Deploy Desktop Applications on the Web

John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch
Microsoft Research

Abstract

Xax is a browser plugin model that enables developers to leverage existing tools, libraries, and entire programs to deliver feature-rich applications on the web. Xax employs a novel combination of mechanisms that collectively provide security, OS-independence, performance, and support for legacy code. These mechanisms include memory-isolated native code execution behind a narrow syscall interface, an abstraction layer that provides a consistent binary interface across operating systems, system services via hooks to existing browser mechanisms, and lightweight modifications to existing tool chains and code bases. We demonstrate a variety of applications and libraries from existing code bases, in several languages, produced with various tool chains, running in multiple browsers on multiple operating systems. With roughly two person-weeks of effort, we ported 3.3 million lines of code to Xax, including a PDF viewer, a Python interpreter, a speech synthesizer, and an OpenGL pipeline.

1 Introduction

Web applications are undergoing a rapid evolution in functionality. Whereas they were once merely simple dynamic enhancements to otherwise-static web pages, modern web apps[1] are driving toward the power of fully functional desktop applications such as email clients (Gmail, Hotmail, Outlook Web Access) and productivity apps (Google Docs). Web applications offer two significant advantages over desktop apps: security—in that the user's system is protected from buggy or malicious applications—and OS-independence. Both of these properties are normally provided by a virtual execution environment that implements a type-safe language, such as JavaScript, Flash, or Silverlight. However, this mechanism inherently prohibits the use of non-type-safe legacy code. Since the vast majority of extant desktop applications and libraries are not written in a type-safe language, this enormous code base is currently unavailable to the developers of web applications.

Our vision is to deliver feature-rich, desktop-class applications on the web. We believe that the fastest and easiest way to create such apps is to leverage the existing code bases of desktop applications and libraries, thereby exploiting the years of design, development, and debugging effort that have gone into them. This existing code commonly expects to run in an OS process and to access OS services. However, actually running the code in an OS process would defeat the OS-independence required by web apps; and it would also impede code security, because large and complex OS system-call (or *syscall*) interfaces are difficult to secure against privilege-escalation vulnerabilities [15].

There is thus a trade-off between **OS-independence**, **security**, and **legacy support**. No existing web-app mechanism can provide all three of these properties. Herein, we show that is possible to achieve all three, and further to achieve native-code **performance**.

In particular, we propose eliminating the process's access to the operating system, and instead providing only a very narrow syscall interface. A sufficiently narrow interface is easy to implement identically on different operating systems; and it is far easier to secure against malicious code. However, it may not be obvious that a process can do much useful work without an operating system to call, and it is even less clear that legacy code could easily be made to work without an OS.

Surprisingly, we found that with minimal modification, legacy libraries and applications with large code bases and rich functionality can indeed be compiled to run on a very simple syscall interface. We demonstrate this point by running the GhostScript PDF viewer, the eSpeak speech synthesizer, and an OpenGL demo that renders 3D animation. In total, it took roughly two person-weeks of effort to port 3.3 million lines of code to use this simple interface. This existing code was written in several languages and produced with various tool chains, and it runs in multiple browsers on multiple operating systems.

We achieved these results with *Xax*, a browser plugin model that supports legacy code in a secure and OS-independent manner, and which further provides native-code performance as required by feature-rich applications. Xax achieves these properties with four mechanisms:

- The *picoprocess*, a native-code execution abstraction that is secured via hardware memory isolation and a very narrow system-call interface, akin to a streamlined hardware virtual machine
- The Platform Abstraction Layer (PAL), which provides an OS-independent Application Binary Interface (ABI) to Xax picoprocesses
- Hooks to existing browser mechanisms to provide applications with system services—such as network communication, user interface, and local storage—that respect browser security policies
- Lightweight modifications to existing tool chains and code bases, for retargeting legacy code to the Xax picoprocess environment

The key principle behind Xax is that the browser already contains sufficient functionality to support the necessary system services for running legacy code. Xax provides this support with its novel combination of four mechanisms and its specific design decisions within each mechanism. Together, these choices achieve our goal of high-performance support for legacy desktop code in secure, OS-independent web applications.

Xax provides key pieces of a comprehensive solution to enable skilled developers to deploy actual desktop applications on the web. Although we have not yet built a large, full-featured application, we have built several moderate-sized applications using a dozen libraries and application components we have ported.

In addition, by leveraging not only existing application code and libraries but also existing development tool chains, Xax allows even moderately skilled developers to combine the conventional DOM-manipulation model of web applications with the power of existing non-web-specific code libraries, arbitrary programming languages, and familiar development tools. We demonstrate this by porting a Python interpreter to Xax and providing language bindings to JavaScript DOM functions, after which we created a social-network visualization app using unmodified Python wrappers for the graphviz library.

Finally, we show that the Xax plugin model can actually subsume other browser plugins. We demonstrate this with a basic port of the Kaffe Java Virtual Machine (JVM) into Xax. Because Kaffe runs within a Xax picoprocess, it does not add to the browser's trusted code base, unlike the standard JVM browser plugin.

The next section details the goals of Xax and contrasts with alternative approaches. Section 3 describes the four mechanisms Xax uses to achieve its goals: picoprocesses, the Platform Abstraction Layer, services via browser mechanisms, and lightweight code modification. Section 4 describes our implementations of Xax in Linux and Windows, as well as our proxy-based browser integration. Section 5 describes some of our example applications. Section 6 evaluates the four benefits of Xax described in Section 2. Sections 7 and 8 describe related and future work. Section 9 summarizes and concludes.

2 Goals and Alternatives

In this section, we detail the goals that must be satisfied to deliver desktop applications on the web, and we consider alternative mechanisms for achieving these goals.

2.1 Xax Design Goals

As previewed in the Introduction, Xax has four design goals: security, OS-independence, performance, and legacy support. For the first three, our intent is to match the benefits of existing web-app mechanisms, such as JavaScript and Flash. Xax's main benefit beyond existing mechanisms is support for legacy code.

security — The particular form of security required for web applications is protecting the client against malicious code. (For the scope of this paper, we ignore other threats such as cross-site scripting and phishing.) Part of what makes web applications attractive is that they are supposed to run safely without requiring explicit trust assumptions from the user. This stands in contrast to installed desktop applications, which have nearly unfettered access to the client machine, so users make trust assumptions whenever they install a desktop program from a CD or via the Internet. Web applications are considered safe because they execute within a sandbox that sharply restricts the reach of the program.

OS-independence — Unlike desktop apps, web applications are not tied to a particular operating system, because they do not make direct use of OS services. Instead, web apps invoke services provided by the browser or by a browser plugin, which is responsible for exporting the same interface and semantics across OS implementations. Ideally, web apps are also independent of the particular browser in which they run; however, some aspects of HTML and the JavaScript environment are not implemented consistently among browsers [22], which somewhat limits this benefit in practice. In addition, a particular web app might rely on features of a particular plugin version, so running the web app might require downloading and installing a new version of the plugin (which entails making a trust assumption).

performance — Simple web apps, such as web pages with dynamic menus, may not require much performance from their execution environments. However, the performance demands may be significant for feature-rich applications that provide functionality comparable to desktop applications, such as animated 3D rendering.

legacy support — Developing complex, feature-rich applications requires an enormous effort. A nearly essential practice for mitigating this effort is software reuse, which has been a staple of the computer industry since the idea was first proposed in 1969 [27]. Despite the fact that the past decade has seen an increasing amount of new code written in type-safe languages, the vast majority of extant software is not type-safe. Of 311 million lines of code in the SourceForge [40] repository, half are C (29%) and C++ (18%); by contrast, Java, C# and JavaScript combined account for only 17%. Large fractions of the 86 million lines [21] of Mac OS X, 200 million lines [1] of Windows Vista, and 283 million lines [36] of the Debian GNU/Linux distribution are general libraries that provide significant functionality to desktop applications; much of this legacy code could benefit the development of rich web applications.

2.2 Alternative Mechanisms

Xax achieves all four of the above goals. Many existing mechanisms for developing web apps already exist, but each falls short of these goals in at least some respects. There are also other yet-undeployed approaches one could explore; we argue that Xax has advantages over each of these other approaches.

2.2.1 Existing web-app mechanisms

There are a number of existing mechanisms for implementing web applications. This set of mechanisms cannot be totally ordered with respect to our four goals; however, we make an attempt to present them in roughly increasing order of goal satisfaction.

JavaScript is an interpreted scripting language with dynamic typing and very late binding. It is included in all major web browsers, so it has the exclusive benefit of not requiring a plugin. It provides language-based security and OS-independence. Because it is interpreted, it does not have very good performance, and the late-binding dynamic semantics of the language make it difficult to JIT and therefore slow. Conversion tools provide limited legacy support for other type-safe languages, including Python, Java, C#, and Pascal, but not for any non-type-safe language.

ActiveX controls are means for packaging client-side code that can be invoked from a web page. They execute natively, so they provide high performance. They have some legacy support for non-type-safe code, particularly for C++ code that is compiled with the Microsoft Foundation Class (MFC) library or the Active Template Library (ATL), as well as support for other languages such as Delphi and Visual Basic. ActiveX controls work only on Windows, and because they have unrestricted OS access, they provide no security against malicious code.

Type-safe intermediate-language systems include Flash, the Java Virtual Machine (JVM), and Silverlight. These all provide security via translating a type-safe source language into a type-safe intermediate language—such as bytecode—that is downloaded to the browser. The definition of the intermediate language is OS-independent, and interpreters or JIT compilers exist for all major browsers. Performance is good because of JIT compiling. Collectively, these systems support a sizeable count of type-safe languages: Flash bytecode can be generated from ActionScript and LZX. Java bytecode can be generated from Java, Python, Ruby, JavaScript, and Common Lisp. Common Language Runtime code, Silverlight 2's intermediate language, can be generated from C#, Visual Basic, Managed C++, and a number of uncommon languages. However, none of these systems can support legacy code written in a non-type-safe language, which — as observed above — is the vast majority of extant code.

2.2.2 OS processes

Since the lion's share of legacy code was written to run in an OS process and to access OS services, a natural way to support this code within a web application is to actually run it inside an OS process. This approach provides the performance of native-code execution as well as direct legacy support. However, it leads to two problems that could potentially be solved at some cost.

First, and most obviously, OS processes are not OS-independent. However, it is possible to write compatibility layers [47] that that map foreign OS calls to native OS calls. Such compatibility layers are notoriously hard to write, because OS processes require bug-for-bug binary-compatible emulation of the OS interface.

Second, OS processes provide insufficient safety for web apps, since the interface to the OS is powerful enough for the process to harm the client machine. However, it is possible to write confinement layers [16, 17, 34] that restrict the allowable system calls made by a process. Such confinement layers are also quite challenging to create, not because the mechanism is particularly difficult, but because of the subtleties in defining appropriate policies that are sufficiently liberal to permit application functionality while sufficiently restrictive to prevent security breaches [15].

More broadly, we believe that trying to pare away dangerous entry points and combinations of calling parameters from a wide and complex interface is fraught with error. As described below, Xax takes the opposite tack by starting with no interface and then adding the minimum necessary to provide useful functionality.

2.2.3 Hardware virtual machines

Another alternative is to run a legacy application, along with the OS for which it was written, inside a hardware virtual machine (VM). Modern VM technology provides impressive performance that rivals native execution speed. It achieves strong security against malicious code through a combination of isolation via hardware and communication via a virtual network interface. A VM that runs on top of multiple host OSes can provide OS-independence, and a VM that executes multiple guest OSes can provide full legacy support. Tahoma contains web browsers using VMs [8]. However, use of a VM for executing web applications leads to three concerns.

First, the virtual machine monitor (VMM), which provides the hardware emulation environment for a VM, is part of the trusted code base. Because VMMs are large and complex, they contain significant potential for security vulnerabilities [32].

Second, virtual machine images are very large, because they include not only the application and libraries but also a full OS. For instance, we measured that a sparsely configured Debian system fetches over 25 MB of pages merely to boot. Given typical wide-area connection speeds, VM images can take hours to download, even with optimizations [23, 5, 39]. It is plausible that sophisticated caching and prefetching strategies could mitigate some of this download time. In addition, the techniques we use to port apps to Xax (§3.4) could similarly be applied to reducing VM image sizes.

Third, VM technology is challenging to implement, which makes it difficult to extend to other platforms, such as mobile devices. A VMM must perform accurate emulation of kernel-mode execution, a full MMU, addressable and programmable devices, and the convoluted addressing modes employed during OS boot. This complexity is so challenging that only one fully virtualizing VM product is currently able to run multiple guest OSes and to run on multiple host OSes [45].

The complexity of VM technology can be reduced by paravirtualization [46], which entails making small changes to the guest code to reduce the emulation burden on the VM system. Such changes may obviate some of the more cumbersome addressing modes or eliminate the need for binary rewriting to mask unvirtualizable machine features.

We observe that the paravirtualization concept can be taken to an extreme. Rather than merely modifying the guest OS, one can eliminate the guest OS along with some of the guest libraries, and then make changes to the guest application that enable it to run on a dramatically less-functional substrate. Such an approach substantially reduces the size and complexity of the VMM, since it need not emulate physical devices, the MMU, or CPU kernel mode. In addition, this approach dramatically reduces the size of VM images, since they contain little more than application code. Moreover, reducing the size of the VMM reduces the size of the trusted code base, thereby improving security. Such extreme paravirtualization is one way to view the Xax picoprocess architecture, which we describe next.

3 Mechanisms

Section 2.1 itemized four design goals, and the present section describes the four mechanisms by which Xax achieves these goals. Despite the agreement in number, there is not a one-to-one correspondence between the goals and the mechanisms. Security is provided by picoprocesses and browser-based services; OS-independence is provided by picoprocesses and the Platform Abstraction Layer; performance is provided by picoprocesses; and legacy support is provided by lightweight code modification.

3.1 Picoprocesses

The core abstraction in Xax is the *picoprocess*. As described in the previous section, a picoprocess can be thought of as a stripped-down virtual machine without emulated physical devices, MMU, or CPU kernel mode. Alternatively, a picoprocess can be thought of as a highly restricted OS process that is prevented from making kernel calls. In either view, a picoprocess is a single hardware-memory-isolated address space with strictly user-mode CPU execution and a very narrow interface to the world outside the picoprocess, as illustrated in Figure 1.

Picoprocesses are created and mediated by a browser plugin called the *Xax Monitor*. Like a virtual machine monitor (in the VM analogy) or an OS kernel (in the OS process analogy), the Xax Monitor is part of the browser's trusted code base, so it is important to keep it small. The picoprocess communicates by making *xaxcalls* (analogous to syscalls) to the Xax Monitor.

Because the Xax Monitor uses OS services to create and manage picoprocesses, it is necessarily OS-specific. Moreover, to ease the implementation burden and help keep the Xax Monitor simple, we do not enforce a standard xaxcall interface. The specific set of xaxcalls, as well as the xaxcall invocation mechanism, may vary depending on the underlying OS platform. We describe some differences below in sections on our Linux (§4.2) and Windows (§4.3) implementations. In terms of functionality, xaxcalls provide means for memory allocation and deallocation, raw communication with the browser, raw communication with the origin server, access to URL query parameters, and picoprocess exit.

The simplicity of the xaxcall interface makes it very easy to implement on commodity operating systems, which assists OS-independence. This simplicity also

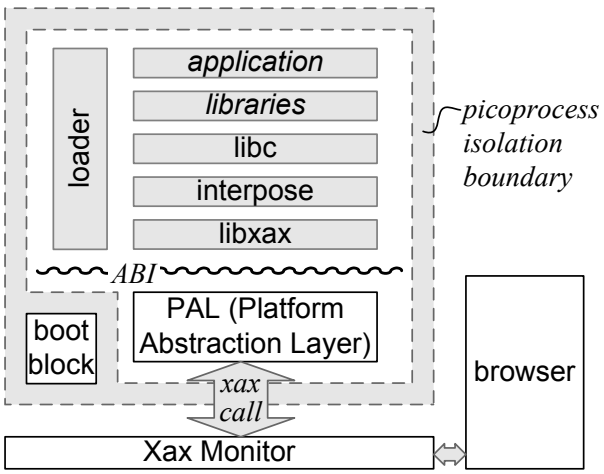


Figure 1: The Xax architecture. Everything inside the picoprocess (§3.1) isolation boundary is untrusted. The Xax Monitor mediates access to the outside world, employing existing browser mechanisms (§3.3) to implement xaxcalls from the picoprocess. The PAL (§3.2) provides a consistent Application Binary Interface (ABI) across OS platforms. Above the ABI, the specific structure can vary; the depicted structure is one we have found useful when porting code (§3.4).

aids security, since it is much easier to reason about the security aspects of a narrow interface with simple semantics than a wide interface with complex semantics. Because a picoprocess executes native code, it provides good performance. However, it is not necessarily clear that this architecture supports legacy code that was written with the expectation of running in an OS process with access to rich OS services; we address this point in §3.4 below.

3.2 Platform Abstraction Layer

As mentioned in the previous section, the xaxcall interface may vary slightly across OS platforms. For OS-independence, Xax defines a consistent *Application Binary Interface* (ABI) irrespective of the underlying OS. By necessity, the ABI varies across architectures, so the x86 ABI is different from the PowerPC ABI.

The ABI is exported by an OS-specific *Platform Abstraction Layer* (PAL), which translates the OS-independent ABI into the OS-specific xaxcalls of the Xax Monitor. The PAL is included with the OS-specific Xax implementation; everything above the ABI is native code delivered from an origin server. The PAL runs inside the Xax picoprocess, so its code is not trusted. Security is provided by the xaxcall interface (dashed border in Figure 1); the PAL merely provides ABI consistency across host operating systems (wiggly line in Figure 1).

All xaxcalls are nonblocking except for `poll`, which can optionally yield until I/O is ready. This provides sufficient functionality for user-level threading.

We herewith present the entire Xax ABI. For memory allocation and deallocation, the ABI includes the following two calls:

```
void *xabi_alloc(
  void *start, long len);
```

Map `len` zero-filled bytes of picoprocess memory, starting at `start` if specified. Return the address.

```
int xabi_free(void *start);
```

Free the memory region beginning at `start`, which must be an address returned from `xabi_alloc`. Return 0 for success or -1 for error.

As described in the next section, the picoprocess appears to the browser as a web server, and communication is typically over HTTP. When the browser opens a connection to the picoprocess, this connection can be received by the following call:

```
int xabi_accept();
```

Return a channel identifier, analogous to a Unix file descriptor or a Windows handle, connected to an incoming connection from the browser. Return -1 if no incoming connection is ready.

The picoprocess can also initiate connection to the server that provided the picoprocess application. To initiate a connection to the home server, the picoprocess uses the following call:

```
int xabi_open_url(
  const char *method,
  const char *url);
```

Return a channel identifier connected to the given URL, according to the specified `method`, which may be "get", "put", or "connect". Fetch and cache the URL according to the Same Origin Policy (SOP) rules for the domain that provided the Xax picoprocess.

The operations that can be performed on an open channel are `read`, `write`, `poll`, and `close`:

```
int xabi_read(
  int chnl, char *buf, int len);
int xabi_write(
  int chnl, char *buf, int len);
```

Transfer data on an open channel. Return the number of bytes transferred, 0 if the channel is not ready, or -1 if the channel is closed or failed.

```
typedef struct {
  int channel;
  short events;  /* requested */
  short revents; /* returned */
} xabi_poll_fd;
```

```
int xabi_poll(
  xabi_poll_fd *pfds, int npfds,
  bool block);
```
Indicate the ready status of a set of channels by updating `revents`. If `block` is true, do not return until at least one requested event is ready, thereby allowing the picoprocess to yield the processor. Return the number of events ready; do not return 0 if `block` is true.

```
int xabi_close(int chnl);
```
Close an open channel. Return 0 for success or -1 for error.

During picoprocess boot, the loader (§4.4) needs to know the URL from which to fetch the application image. We could have required a custom loader for each application, with the URL baked into the loader's image. Instead, we wrote a general loader that reads the application URL from the query parameters of the URL that launched the picoprocess. The following call, which in normally used only by the loader, provides access to these parameters. (Note that there is no corresponding xaxcall; the parameters are written into the PAL during picoprocess initialization.)

```
const char **xabi_args();
```
Return a pointer to a NULL-terminated list of pointers to arguments specified at instantiation.

Lastly, the ABI provides a call to exit the picoprocess when it is finished:

```
void xabi_exit();
```

Although the PAL runs inside the picoprocess, it is not part of the application. More pointedly, it is not delivered with the OS-independent application code. Instead, the appropriate OS-specific PAL remains resident on the client machine, along with the Xax Monitor and the web browser, whose implementations are also OS-specific. When a Xax application is delivered to the client, the app and the PAL are loaded into the picoprocess and linked via a simple dynamic-linking mechanism: The ABI defines a table of function pointers and the calling convention for the functions. For x86 architectures, this calling convention is `cdecl`; for the PowerPC, it is the one standard calling convention; and, for other architectures, no Xax ABI has yet been defined.

We have found it helpful to create a simple shim library called `libxax` that an application may statically link. `libxax` exports a set of symbols (`xabi_read`, `xabi_open_url`, etc.) that obey the function linkage convention of the developer's tool chain. The shim converts each of these calls to the corresponding ABI call in the PAL. This shim thus provides a standard Application Programming Interface (API) to Xax applications.

3.3 Services via browser mechanisms

A key Xax principle is that there is sufficient functionality within the browser to support the system services needed by web applications. In fact, we assert not only that it is sufficient for the Xax Monitor to employ the browser's functionality, but also that doing so improves the system's security. Because Xax reuses the existing security policy—and much of the mechanism—in the browser, Xax introduces no new security vulnerabilities, modulo implementation bugs in the Xax Monitor's (small) trusted code base.

The Xax Monitor has the job of providing the services indicated by the xaxcall interface. Some of these services are straightforward for the Xax Monitor to perform directly, such as memory allocation/deallocation, access to URL query parameters, and picoprocess exit. The Xax Monitor also provides a communication path to the browser, via which the Xax picoprocess appears as a web server. This communication path enables the Xax application to use `read` and `write` calls to serve HTTP to the browser. From the browser's perspective, these HTTP responses appear to come from the remote server that supplied the Xax app. It is clear that this approach is secure, since the Xax application is unable to do anything that the remote server could not have done by serving content directly over the Internet.

Using the picoprocess-to-browser communication path, the Xax application can employ JavaScript code in the browser to perform functions on its behalf, such as user interface operations, DOM manipulation, and access to browser cookies. In our applications, we have applied a common design pattern: The Xax app provides an HTML page to the browser, and this page contains JavaScript stubs which translate messages from the picoprocess into JavaScript function invocations.

It would be possible but awkward to use JavaScript for network communication. To pass through JavaScript, an application or library binary from a remote server would have to be uuencoded, encapsulated in JSON, transferred via HTTP, de-encapsulated, and decoded. To simplify this process, we provide the ABI call `xabi_open_url` to allow direct communication between a Xax picoprocess and its origin server. Both our Linux and Windows Xax Monitors provide corresponding xaxcalls that implement the primitives efficiently.

3.4 Lightweight code modification

One of our most surprising findings is how little effort it takes to port a legacy application, library, or tool chain to the minimalist Xax ABI. This is surprising because this legacy code was written to run atop an operating system, and it was not *a priori* obvious that we could eliminate the OS and still enable the legacy code to perform its main function. For instance, to enable development of

the app described in §5.2, we ported the graphviz library and the Python interpreter to Xax. Using `strace`, we saw that a quick test application makes 2725 syscalls (39 unique). Porting this code to Xax would seem to require an enormous emulation of OS functionality. However, using our lightweight modifications, we ported this million lines of code in just a few days.

Although the particular modifications required are application-dependent, they follow a design pattern that covers five common aspects: disabling irrelevant dependencies, restricting application interface usage, applying failure-oblivious computing techniques, internally emulating syscall functionality, and (only when necessary) providing real syscall functionality via xaxcalls.

The first step is to use compiler flags to disable dependencies on irrelevant components. Not all libraries and code components are necessary for use within the web-application framework, and removing them reduces the download size of the web app and also reduces the total amount of code that needs to be ported. For Python/graphviz, by disabling components such as pango and pthreads, we eliminated 699 syscalls (16 unique).

The second step is to restrict the interfaces that the application uses. For instance, an app might handle I/O either via named files or via `stdin`/`stdout`, and the latter may require less support from the system. Depending on the app, restricting the interface is done in various ways, such as by setting command-line arguments or environment variables. For Python/graphviz, we used an entry-point parameter to change the output method from "xlib" to "svg", which eliminated 367 syscalls (21 unique).

The third step is to identify which of the application's remaining system calls can be handled trivially. For example, we can often return error codes indicating failure, in a manner similar to failure-oblivious computing [35]. For Python/graphviz, it was sufficient to simply reject 125 syscalls (11 unique). Specifically, we obviate `getuid32`, `rt_sigaction`, `fstat64`, `rt_sigprocmask`, `ioctl`, `uname`, `gettimeofday`, `connect`, `time`, `fcntl64`, and `socket`.

The fourth step is to emulate syscall functionality within the syscall interpose layer (see Figure 1). For instance, Python/graphviz reads Python library files from a file system at runtime. We package these library files as a tar ball, and we emulate a subset of file-system calls using libtar to access the libraries. The tar ball is read-only, which is all Python/graphviz requires. For some of our other ported applications, we also provide read/write access to temporary files by creating a RAM disk in the interpose layer. Code in the interpose layer looks at the file path to determine whether to direct calls to the tar ball, to the RAM disk, or to somewhere else, such as a file downloaded from the origin server. For Python/graphviz, we use internal emulation to satisfy 1409 syscalls (14 unique), 943 of which fail obliviously.

The fifth and final step is to provide real backing functionality for the remaining system calls via the Xax ABI. For Python/graphviz, most of the remaining syscalls are for user input and display output, which we route to UI in the browser. We provide this functionality for the remaining 137 syscalls (11 unique). Specifically, we implement `setsockopt`, `listen`, `accept`, `bind`, `read`, `write`, `brk`, `close`, `mmap2`, `old_mmap`, and `munmap`.

The first three steps are application-specific, but for the final two steps, we found much of the syscall support developed for one app to be readily reusable for other apps. For example, we originally wrote the internally emulated tar-based file system to support eSpeak, and we later reused it to support Python. Similarly, the backing functionality for the `mmap` functions and networking functions (`listen`, `accept`, `bind`, ...) is used by all of our example applications.

For any given application, once the needed modifications are understood, the changes become mechanical. Thus, it is fairly straightforward for a developer to maintain both a desktop version and a Xax version of an app, using a configure flag to specify the build target. This is already a common practice for a variety of applications that compile against Linux and BSD and Win32 syscall interfaces.

4 Implementation

In this section, we describe the implementations of Xax on Linux and Windows, as well as our proxy-based browser integration.

Although they have some significant differences, our two implementations of Xax share much common structure. The main aspect in which they differ is in the kernel support for picoprocess isolation and communication, which we will discuss after first describing the common aspects.

4.1 Monitor, boot block, and PAL

The Xax Monitor is a user-mode process that creates, isolates, and manages each picoprocess (§3.1), and that provides the functionality of xaxcalls (§3.3). A picoprocess is realized as a user-level OS process, thus leveraging the hardware memory isolation that the OS already enforces on its processes. Before creating a new picoprocess, the Xax Monitor first allocates a region of shared memory, which will serve as a communication conduit between the picoprocess and the Monitor. Then, the picoprocess is created as a child process of the Xax Monitor process.

This child process begins by executing an OS-specific *boot block*, which performs three steps. First, it maps the shared memory region into the child process's address space, thereby completing the communication conduit. Second, it makes an OS-specific kernel call that permanently revokes the child process's ability to make subsequent kernel calls, thereby completing the isolation. Third, it passes execution to the OS-specific PAL, which in turn loads and passes execution to the Xax application.

Note that the boot block is part of the TCB, even though it executes inside the child process. The child process does not truly become a picoprocess until after the boot block has executed. At that point, the child process has no means to de-isolate itself, since this would require a kernel call but the picoprocess is prevented from making kernel calls.

After transferring control to the Xax application, the PAL (§3.2) has the job of implementing the Xax ABI by making appropriate xaxcalls to the Xax Monitor. To make a xaxcall, the PAL writes the xaxcall identifier and arguments into the shared memory region, then traps to the kernel. In an OS-specific manner (described below) the kernel notifies the Xax Monitor of the call. The Monitor then reads the shared memory, performs the indicated operation, writes the result to the shared memory, and returns control to the picoprocess.

Although the Xax Monitor has different implementations on different operating systems, it handles most xaxcalls in more-or-less the same way irrespective of OS. The `alloc` and `free` xaxcalls are exceptions to this rule, so their different implementations are described in the following two sections. For `accept`, the Xax Monitor maintains a queue of connection requests from the browser, and each call dequeues the next request. The `open_url` xaxcall makes an HTTP connection to a remote resource; the returned channel identifier corresponds to either a socket handle or a file handle, depending on whether the requested data is cached. The I/O calls `read`, `write`, `poll`, and `close` are implemented by reading, writing, polling, and closing OS file descriptors on sockets and files. The `exit` xaxcall simply terminates the child process.

4.2 Linux kernel support

Our Linux implementation involves no custom kernel code. Instead, it makes use of the Linux kernel's `ptrace` facility, which enables a process to observe and control the execution of another process.

As described above, the boot block makes a kernel call to revoke the child process's ability to make subsequent kernel calls. In our Linux implementation, this is done by calling `ptrace(TRACE_ME)`, which causes the kernel to intercept the entry and exit of every subsequent syscall, transferring control to the Xax Monitor parent process. On entry to a syscall, the Xax Monitor normally replaces whatever system call the child process requested with a harmless system call (specifically, `getpid`) before releasing control to the kernel. This prevents the child process from passing a syscall to the OS.

Syscalls are also legitimately used by the PAL to signal a xaxcall. Thus, when `ptrace` notifies the Xax Monitor of an entry to a syscall, the Monitor checks whether the shared memory contains a legitimate xaxcall identifier and arguments. If it does, the Xax Monitor performs the operation and returns the result, as described above. If the xaxcall is a memory-management operation (`alloc` or `free`), it has to be handled specially, because Linux does not provide a mechanism for a process to allocate memory on behalf of another process. So, in this case, the Xax Monitor does not overwrite the syscall with `getpid`. Instead, it overwrites the syscall with `mmap` and a set of appropriate arguments. Since the return from the syscall is also intercepted by `ptrace`, the Xax Monitor has an opportunity to write a return value for the `alloc` xaxcall into the shared memory, based on the return value from the `mmap` syscall.

Use of an existing kernel facility (`ptrace`) enables our Linux implementation to be deployed without kernel-module installation or root privilege. However, it entails a performance hit, because every xaxcall requires three syscalls from the Xax Monitor: one to swap out the syscall with `getpid` or `mmap`, a second to enter the kernel, and a third to resume the picoprocess. More importantly, if the Xax Monitor fails and exits without proper signal handling, the child process may continue to run without having its syscalls intercepted [34]. This failure condition could turn the picoprocess back into a regular OS process, which would violate security.

These performance and security problems could be mitigated by using a custom kernel module instead of `ptrace`. In the future, we intend to employ this approach, and we have already done so in our Windows implementation.

4.3 Windows kernel support

In our Windows implementation, when the child process's boot block makes a kernel call to establish an interposition on all subsequent syscalls, it makes this call to a custom kernel module, *XaxDrv*. Because every Windows thread has its own pointer to a table of system call handlers, XaxDrv is able to isolate a picoprocess by replacing the handler table for that process's thread. The replacement table converts every user-mode syscall into an inter-process call (IPC) to the user-space Xax Monitor. For a syscall originating from kernel mode (e.g., for paging), XaxDrv passes the call through to the original handler, preserving the dispatcher's stack frame for the callee's inspection.

When the Xax Monitor receives an IPC, it reads the xaxcall identifier and arguments from the shared memory and performs the operation. Unlike the Linux case, no special handling is required for memory-management operations, because Windows `NtMapViewOfSection` allows the Monitor to map memory on behalf of its child process.

Although XaxDrv has to be ported to each version of Windows on which it runs, the changes are minimal, involving two constant scalars and a constant array: (1) the offset in the kernel thread block for the pointer to the syscall handler table, (2) the count of system calls, and (3) for each system call, the total parameter byte count. This information is readily available from the kernel debugger in the Windows Driver Kit [28]. We have ported XaxDrv to Windows XP, Windows Vista, and Windows Server 2008.

An alternative implementation of XaxDrv could have followed the common approach [25, 38] of patching every entry in the standard system-call table. However, Microsoft discourages this practice because it transparently changes the behavior of every process in the system. Furthermore, even if the interposed handlers were to properly fall through to the original handlers, they would still add overhead to every system call.

4.4 Loaders

The Linux toolchain emits standard statically-linked Elf binaries. These Xax binaries are loaded by a small `elfLoader`. This loader reads the target binary, parses it to learn where to map its program regions, and looks up two symbols: a global symbol where the binary's copy of `libxax` expects to find a pointer to the PAL's dispatch table, and the address of the `_start` symbol. Then `elfLoader` maps the program, writes the dispatch table location into the pointer, and jumps to `_start`.

The Windows toolchain emits statically-linked `.EXE` binaries in Windows' native PE-COFF format. Our `peLoader` performs the corresponding tasks to map and launch PE executables.

4.5 Browser integration

Recall (§3.3) that the Xax application appears to the browser as part of the origin server that just happens to handle HTTP requests very quickly; this ensures that the picoprocess is governed by the Same-Origin Policy [20] just as is the origin server.

Our implementation integrates Xax into the browser via an HTTP proxy. This approach is expedient, and one implementation serves all makes of browser. The proxy passes most HTTP requests transparently to the specified host. However, if the URL's path component begins with `/_xax/`, the proxy interposes on the request to direct the request to an existing picoprocess or to create a new one. The proxy is integrated with the Xax Monitor process, and allows each picoprocess to contact its origin server via `xax_open_url`. This contact employs the same mechanism that fetches ordinary URLs, and thus obeys the SOP.

5 Examples

This section highlights several features of Xax by way of brief presentations of some application examples.

5.1 Headline Reader and 3D Demo

The only connection between a Xax picoprocess and the browser is an HTTP channel, which might seem insufficient to deliver the rich content that can be provided by other plugins. However, we present two applications that show this channel to be sufficient.

First, the Headline Reader app performs text-to-speech conversion. We ported the 25K-line eSpeak speech synthesizer to Xax and invoke it with a small wrapper app we wrote. The app produces .WAV audio clips, which are transferred to the browser via the HTTP channel and then played using the browser's standard audio helper.

Second, the 3D Demo performs real-time 3D rendering. We ported the 684K-line Mesa OpenGL library to Xax. It includes a demo which draws a 400×400-pixel 3D scene; we modified it to animate. We express the output of OpenGL as a series of PNG files, which are sequentially transferred to the browser periodically and inserted into an HTML DIV element for display. This approach is performance-limited by the time spent encoding PNG files; the Xax mesa demo renders 8.8 frames per second on a machine where native OpenGL renders the same scene at 36 frames per second.

5.2 Social Network Visualizer

The lightweight code modifications described in Section 3.4 are not very time-consuming, but they do require a fair degree of sophistication from the developer porting the code. However, we present an application that shows how Xax enables developers with no special skill to create new and interesting apps.

We separately ported a Python interpreter and the graphviz graph-layout library to Xax. We also wrote language bindings between Python and the DOM-manipulation functions in JavaScript, which allows Python code to directly manipulate the DOM. Because there are Python wrappers for graphviz, it is possible for Python code to call the powerful graph visualization routines in this library.

Another developer, who had no familiarity with the process of porting code to Xax, then wrote a web app in Python for visualizing a social network, specifically a network of actors from the Internet Movie DataBase (IMDB), akin to "six degrees of Kevin Bacon". The app's web page provides a text box for entering an ac-

tor's name. The client queries a back-end service to enumerate the movies featuring that actor, as well as lists of other actors in each movie. The client-side web app uses graphviz to plot and present the network of actors and movies.

Xax enabled the developer to leverage the advanced non-type-safe legacy code base of grahpviz, which was developed over more than a decade. It also enabled this developer to use his knowledge of Python to write an app without needing to learn a new language, such as JavaScript. None of this required the developer to learn how to port code to Xax. The tool and library were ported once, and they are now usable by non-experts.

5.3 GhostScript and Kaffe

The fact that Xax is yet another browser plugin might give one cause for concern. Even though Xax provides benefits unavailable in any existing plugin, a user might still be bothered by having to install an additional plugin in the browser. However, we present two examples that show how the Xax plugin model can actually subsume other browser plugins.

First, we ported the GhostScript PDF viewer to Xax. PDF viewers are currently available as browser plugins, to enable users to view PDF documents on web sites. Xax enables PDF viewing functionality without the need for a special-purpose plugin. Moreover, by running the PDF viewer inside a secure Xax picoprocess, we protect the browser from the dozens of vulnerabilities that have been discovered in PDF plugins [32].

Second, we performed a basic port of the Kaffe Java Virtual Machine (JVM) into Xax. As described in Section 2.2.1, JVM is an alternative mechanism for writing web applications, albeit one that does not provide legacy support for non-type-safe code. Although we have not yet completed our port of the Qt implementation of Kaffe's Abstract Windowing Toolkit, the core Java execution engine is working and able to perform UI functions via DOM manipulation. As with the PDF renderer, by running the JVM inside a picoprocess, we protect the browser from vulnerabilities in the JVM implementation [4, 9, 10, 37].

6 Evaluation

Here we evaluate Xax's performance, legacy support, OS-independence, and security.

6.1 Performance

To evaluate performance, we run microbenchmarks and macrobenchmarks to measure CPU- and I/O-bound performance. All measurements are on a 2.8GHz Intel Pentium 4.

Xax's use of native CPU execution, adopted to achieve legacy support, also leads to native CPU performance. Our first microbenchmark (Table 1(a)) computes the SHA-1 hash of H.G. Wells' *The War of the Worlds*. Xax performs comparably to the Linux native host. The Windows native binary was compiled with a different compiler (Visual Studio vs. gcc); we believe this explains the improved performance of the Windows native cases.

Environment	tool	compute `sha1` (a)	syscall `close` (b)	alloc `16MB` (c)
Linux native	gcc	5,930,000	430	27,120
Linux Xax	gcc	5,970,000	69,400	202,600
XP native	VS	4,540,000	1,126	31,390
XP Xax	gcc	6,170,000	16,880	235,300
Vista native	VS	4,580,000	1,316	40,900
Vista Xax	gcc	6,490,000	59,900	612,000

Table 1: Microbenchmarks (§6.1). Units are machine cycles, $1/(2.8 \times 10^9)$ sec; max $\frac{\sigma}{\mu}$=6.6%.

The benefits of native execution led us to accept overheads associated with hardware context switching; however, our simple uninvasive user-level implementations lead to quite high overheads. Table 1(b) reports the cost of a null `xaxcall` compared with a null native system call; in each case, we invoke `close(-1)`. Table 1(c) reports the cost of allocating an empty 16MB memory region. The Xax overhead runs 7–161×.

Despite high `xaxcall` overhead, real applications perform quite well because applications use I/O sparingly. Two macrobenchmarks quantify this observation. First, we wrote a Mandelbrot Set viewer that draws 300 distinct 400×400-pixel frames as it zooms into the Set (Table 2(a)). We chose this application for two reasons. First, it involves both intensive computation and frequent I/O. Second, it is small enough to reimplement in multiple languages, enabling us to compare a single application across multiple extension mechanisms. The Mandelbrot benchmark ran at roughly the same speed under both Windows and Linux, under both Xax and using native binaries. For comparison, we also tried two other common browser extension languages: Java and JavaScript. The benchmark ran 30% faster using Sun Java 1.6r7 under Windows XP. This difference arises from the time taken encoding PNG images in the C implementations; to validate this hypothesis, we removed the PNG step, and found the C implementation as fast as Java. Google Chrome's JavaScript engine (build 2200) required more than an hour. We gave up on Internet Explorer 7 and Firefox 3's JavaScript engines after waiting ten minutes for the first two frames.

Second, we compare the performance of rendering and displaying a one-page Postscript document in Xax and on the native host, using Ghostscript 8.62 as the underlying rendering engine in all cases (Table 2(b)). Ghostscript is a rich application that exercises CPU, allocation, and

Environment	Mandelbrot 300 frames (a)	Ghostscript 1 page (b)
Linux native	169s	840ms
Linux Xax	170s	541ms
Win XP native	188s	835ms
Win XP Xax	177s	738ms
Win XP Java	138s	—
JavaScript	4,870s	—

Table 2: Macrobenchmarks.

I/O. The native benchmarks incur an additional overhead of process launch; this test shows only that the Xax performance is reasonable.

6.2 Legacy Support

To evaluate legacy support, we built Xax applications that use 15 libraries totaling 3.3 million lines of code (LoC) in four languages (Table 3). Only minimal changes were needed to compile the libraries. Most changes were configure flags to specify different library dependencies and to emit a static library as output.

6.3 OS-Independence

To evaluate OS-independence, we ran all of the above applications on our Linux 2.6, Windows XP, Windows Vista, and Windows Server 2008 Xax hosts. Figure 2(a) is a Linux-toolchain program running in Firefox on a Linux-PowerPC host. Figure 2(b) is a Windows-toolchain program running in Firefox on Linux-x86. Figure 2(c) is a Linux-toolchain program running in Firefox on Windows XP. Figure 2(d) is a Linux-toolchain program running in Internet Explorer on Windows Vista. Figure 2(e) is a Linux-toolchain program running in Firefox on Linux-x86.

6.4 Security

We roughly compare the strength of different isolation mechanisms by the count of lines of code in their TCBs [6]. The Xax picoprocess TCB is less than 5,000 lines (See Table 4). In contrast, language-based Flash and Java have implementations around two orders of magnitude bigger. Section 2.2.3 considered the alternative of a hardware VM; note that Xen's TCB is similarly large.

The previous comparison is somewhat generous to Xax, because the table counts Kaffe's entire TCB, including both its isolated execution engine (the JVM, around 50,000 lines) and the new native code Kaffe introduces to provide features like rich GUI displays. On the other hand, as a type-safety-based extension mechanism, Kaffe incorporates native UI code in its TCB for performance or to exploit a stack of legacy code. By contrast, Xax Kaffe isolates any native UI code it incorporates. In future work, we expect Xax to isolate Kaffe's

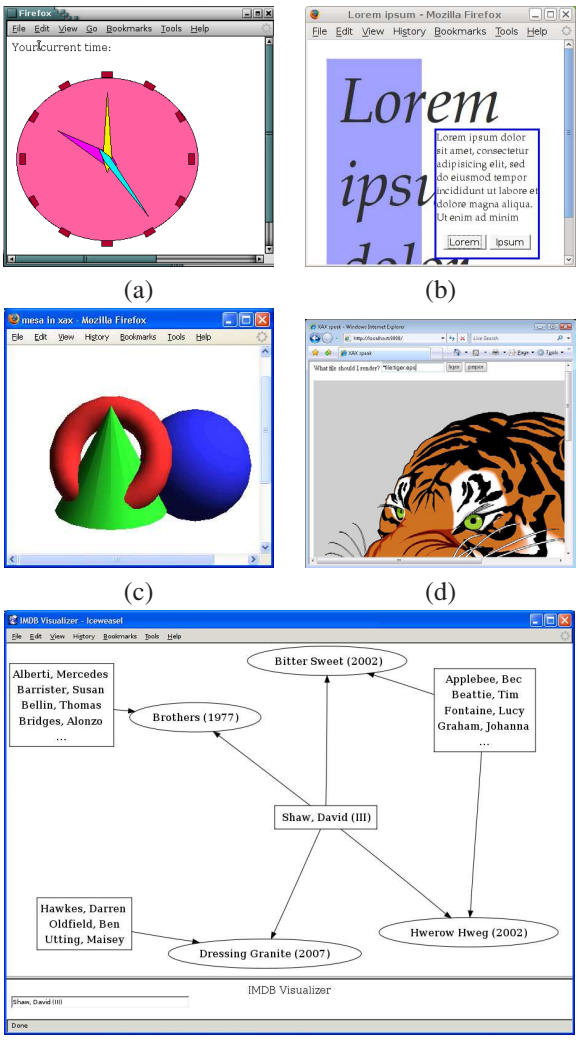


Figure 2: Screenshots of the applications itemized in Table 3.

UI stack with minimal TCB growth. By analogy, note that our PDF viewer isolates a 600K-line component that is otherwise commonly installed in the TCB.

7 Related Work

One of the key observations that enables Xax to achieve its benefits (§2.1) is that today's type-safety-based browser extension mechanisms do not admit legacy code. We chose a simple isolation mechanism buildable from primitives available in commodity operating systems, but several alternative approaches exist.

7.1 Hardware isolation

Other extension systems use memory hardware isolation differently. Nooks isolates drivers from a monolithic kernel [42]. Mondrix has a similar goal, but requires hypothetical hardware support [48]. Palladium isolates kernel extensions by exploiting x86's arcane, otherwise-

Application	Fig.	Language	Toolchain	New LoC	Mods LoC	Base LoC
XaxAnalogClock	2(a)	C	gcc 4.1.2	459		
elfLoader		C	gcc 4.1.2	552		
xaxlib		C	gcc 4.1.2	412		
dietlibc 0.31		C	gcc 4.1.2		405	66,970
zlib 1.2.3		C	gcc 4.1.2		24	25,475
libpng 1.2.25		C	gcc 4.1.2		33	60,925
gd 2.0.35		C	gcc 4.1.2		20	72,202
Kevin Bacon Visualizer	2(e)	Python	python 2.5	143		
xaxFsLib		C	gcc 4.1.2	1,706		
libtar 1.2.11		C	gcc 4.1.2		17	11,348
Python 2.5		C, Python	gcc 4.1.2		33	771,112
Zziplib 0.13.49		C, C++	gcc 4.1.2		40	54,728
Jpeg 6b		C, Asm	gcc 4.1.2		36	26,695
Expat 2.0.1		C	gcc 4.1.2		41	47,813
GraphViz 2.18		C, C++	gcc 4.1.2		79	343,096
Headline Reader		C	gcc 4.1.2	323		
eSpeak 1.36		C++	g++ 4.1.2		46	25,170
3D Demo	2(c)	C, OpenGL	gcc 4.1.2	257		
Mesa 7.0.3		C	gcc 4.1.2		56	683,883
PDF Viewer	2(d)	C	gcc 4.1.2	366		
GhostScript 8.62		C	gcc 4.1.2		29	666,216
Hello Webserver		Java	sun jdk 1.6	67		
Kaffe 1.1.0		C, C++	gcc 4.1.2		123	364,560
Hello Webserver	2(b)	C	Vsl. Studio 2003	189		
peLoader		C	gcc 4.1.2	613		
libc.lib		C			0	57,223
Total					982	3,277,416

Table 3: We have compiled a variety of applications for Xax using a variety of libraries and compiler toolchains comprising over 3.3 million LoC. Each library we build on is listed once, regardless of how many applications use it. For the base libraries, "our LoC" consist mostly of build configuration. LoC is measured by cloc.pl [6].

Isolation mechanism	TCB LoC
Linux `Monitor+Proxy`	2,596
Linux syscall entry path	1,632
	4,228
Windows `Monitor+Proxy`	3,043
Xax kernel driver	978
NT syscall entry path	313
	4,334
Gnash Flash player + deps	791,453
Kaffe non-Java code + deps	280,622
Xen VMM[2]	187,688

Table 4: Security. Because Xax exploits hardware memory protection, both Xax implementations have small TCBs.

unused segment-based isolation mechanism [3]. Each of these mechanisms emphasizes lightweight rich-pointer interactions between a minimally-modified extension and its host environment. Thus the systems improve robustness of existing code compositions, but preventing malice is an explicit non-goal.

7.2 Binary rewriting

Another approach to isolation is binary rewriting. Basic software-fault isolation for RISC [44] and CISC [26] architectures has quite high overhead if required to enforce both read- and write-safety; such mechanisms are thus envisioned as robustness-improving rather than adversary-proof. XFI [12] showed how to achieve adversary-proof isolation for operating system extensions, but overheads still range from 28–116%, and the system has been applied only to short pieces of legacy code.

Vx32 is a general-purpose mechanism that combines segments and binary rewriting [14] to achieve low-overhead adversary-proof protection, and might be a possible alternative to our picoprocesses. However, the use of segment registers places an additional constraint on toolchains. More importantly, it excludes other architectures, which may be important for future mobile devices; Xax already runs on PowerPC (§6.3).

Besides isolation, binary rewriting has also been used to provide transparent cross-architecture portability [2]. It may have applicability in the Xax context: Where web developers have only provided an x86 binary, a non-x86-based host may employ binary rewriting to exchange performance for compatibility.

7.3 Operating system-level virtualization

Another way to isolate untrusted web applications is via *operating system-level virtualization*, wherein an OS provides multiple isolated instances for separate subsystems, each with the same API as the underlying OS. Examples include Solaris zones [33] and FreeBSD jails [41]. The implementation of OS virtualization permeates a monolithic kernel, so its TCB is larger and more amorphous than that of the picoprocess mechanism. Furthermore, the mechanism sacrifices OS-independence and is not supported on many deployed OSes.

7.4 Low-level type safety

Another possible alternative for isolating legacy code is the use of a safety-enforcing typed assembly language (TAL) [29], an instance of a proof-carrying code [31]. TAL is type-safety-based mechanism, similar to those described in §2.2.1, but it is lower-level than JVM or .NET's object-oriented type system. For example, it can enforce safety in polymorphic languages without requiring that objects use vtables. Thus one cannot compile TAL to JVM; a separate TAL runtime must be deployed.

TAL should be easier to target than a higher-level type system; however, no current compilers emit TAL for weakly typed languages such as C. One could perhaps produce such a compiler by modifying the back end of a type-retrofitting C compiler such as CCured [30]. Experience with CCured, however, shows several limitations that restrict its applicability to improving robustness rather than adversary-proofing. First, although some work has been done to verify the safety of CCured's output [19], the CCured compiler must generally be trusted. Second, annotating legacy code to convince CCured to compile it efficiently has proven to be quite labor-intensive [7], and even then rarely eliminates all of the "trusted casts" [19].

7.5 Application context

Many of the papers described above mention potential application to web browsers; however, ours is the first to demonstrate how to enable legacy code to be readily compiled and deployed in the web context. VXA restricts extensions to a narrow interface, and shows applicability to a restricted class of applications (codecs) [13]; one important contribution of the present work is to show how a narrow interface to the existing browser is sufficient to support a much broader range of software (§3.3).

8 Limitations and future work

In this section, we discuss limitations of the current Xax implementation and plans for future enhancements.

8.1 Security analysis

We argue that Xax is secure by its small TCB; however, as a practical matter, our implementations reuse commodity operating systems as substrate. A production implementation deserves a rigorous inspection to ensure both that the kernel syscall dispatch path for a picoprocess is indeed closed, and that no other kernel paths, such as exception handling or memory management, are exploitable by a malicious picoprocess. We should also explore alternative implementations that exclude more host OS code from the TCB, such as a MacOS implementation that uses Mach processes, or a VM-like implementation that completely replaces the processor trap dispatch table for the duration of execution of a picoprocess.

8.2 Rich application enhancements

Rich web applications, Xax or otherwise, will require browser support for efficiently handling large binaries (such as remote differential compression [43]), and support for offline functionality [11, 18]. Because Xax applications access resources via the browser, any browser enhancements that deliver these features are automatically inherited by the Xax environment.

When we port a shared-library loader, Xax can experience further performance improvements from selective preloading [24].

8.3 Improved browser integration

Integrating Xax with the browser using a proxy is expedient, but for several reasons it would be better to directly integrate with the browser. First, rewriting the namespace of the origin server is an abuse of protocol. Instead, the browser should provide an explicit `<embed>` object with which a page can construct and name a picoprocess for further reference. Second, the proxy is unaware of when the browser has navigated away from a page, and when it is thus safe to terminate and reclaim a picoprocess. Third, the proxy cannot operate on `https` connections. For these reasons, we plan to integrate Xax directly into popular browsers.

8.4 Threading

Supporting some threading model is a requirement for targeting general applications. The nonblocking I/O interface (§3.2) is sufficient to implement cooperative user-level threading, such as Kaffe's `jthreads`. Adding to the xaxcall interface a mechanism to deliver an asynchronous signal (e.g., when a poll condition is satisfied) is sufficient to implement preemptive user-level threading. Finally, the xaxcall interface could expose a mechanism for launching additional kernel-level threads to en-

able the picoprocess to exploit a multicore CPU. Each mechanism offers improved application performance in exchange for expanding the Xax Monitor's contribution to the TCB.

8.5 Porting additional libraries

Most of the ports in this paper were built on our modifications to dietlibc. Similarly modifying more mainstream `libc`s, such as the GNU C library and Microsoft Visual Studio's standard libraries, will greatly ease porting of other libraries. One challenge in porting either library is their reliance on x86 segment registers to manage thread-local storage. Because segment registers cannot be assigned in user mode, we must emulate or obviate this functionality.

We also plan to port more interactive code. Our first efforts will be aimed at GUI libraries with few dependencies (e.g. Qt/Embedded). We expect to blit frame buffer regions to the browser; keyboard and mouse events we will capture in JavaScript and send back to Xax.

8.6 Relocatable code

The picoprocess restricts application code to a fixed address range (§3.1). This restriction is an arbitrary intersection of the restrictions imposed by commodity operating systems; we have no guarantee that future Xax implementations will not require further narrowing it. We plan to explore an alternative approach: requiring Xax applications to be relocatable, capable of running on an ABI that makes no guarantee about any particular absolute virtual addresses. One possible limitation with this approach is that it may impede an attempt to import existing binaries directly into Xax.

9 Conclusion

We introduce Xax, a browser plugin model that enables developers to adapt legacy code for use in rich web applications, while maintaining security, performance, and OS-independence.

- Xax's security comes from its use of the picoprocess minimalist isolation boundary and browser-based services; we demonstrate that Xax's TCB is orders of magnitude smaller than alternative approaches.
- Xax's OS-independence comes from its use of picoprocesses and its platform abstraction layer; we demonstrate that Xax applications compiled on any toolchain run on any OS host.
- Xax's performance derives from native code execution in picoprocesses; we measure Xax's compute performance to be comparable with native execution, and that even with quite inefficient I/O performance, Xax delivers compelling whole-application performance.
- Xax's legacy support comes from lightweight code modification; we demonstrate that just a few hundred lines of configuration options are sufficient to port 3.3 million lines of legacy libraries and applications.

Over decades of software development in non-type-safe languages, vast amounts of design, implementation, and testing effort have gone into producing powerful legacy applications. By enabling developers to leverage this prior effort into web applications' deployment and execution model, we anticipate that Xax may change the landscape of web applications.

10 Acknowledgments

The authors thank Jeremy Condit, Chris Hawblitzel, Galen Hunt, Emre Kıcıman, Ed Nightingale, and Helen Wang for enlightening discussions and comments on early drafts of this paper. We also thank the anonymous reviewers and our shepherd, Anthony Joseph, for their suggestions.

References

[1] CARDINAL, C. Live From CES: Hands On With Vista—Vista By The Numbers, A Developer Tells All. Presentation at Gear Live, Jan. 2006.

[2] CHERNOFF, A., AND HOOKWAY, R. DIGITAL FX!32 — running 32-bit x86 applications on Alpha NT. In *Proceedings of the USENIX Windows NT Workshop* (1997), pp. 9–13.

[3] CHIUEH, T., VENKITACHALAM, G., AND PRADHAN, P. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of Symposium on Operating Systems Principles (SOSP)* (1999), pp. 140–153.

[4] CIAC M-060. http://www.ciac.org/.

[5] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proceedings of Networked Systems Design and Implementation (NSDI)* (2005), pp. 273–286.

[6] CLoC. http://cloc.sourceforge.net/.

[7] CONDIT, J. personal communication, 2008.

[8] COX, R. S., GRIBBLE, S. D., LEVY, H. M., AND HANSEN, J. G. A safety-oriented platform for Web applications. In *Proc. Symposium on Security and Privacy* (2006), pp. 350–364.

[9] CVE-2003-0111. http://cve.mitre.org/.

[10] CVE-2007-0043. http://cve.mitre.org/.

[11] Dojo Toolkit. http://dojotoolkit.org/offline.

[12] ERLINGSSON, U., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: software guards for system address spaces. In *Proceedings of Operating Systems Design and Implementation (OSDI)* (2006), pp. 75–88.

[13] FORD, B. VXA: A virtual architecture for durable compressed archives. In *Proceedings of File and Storage Technologies (FAST)* (2005), pp. 295–308.

[14] FORD, B., AND COX, R. Vx32: Lightweight user-level sandboxing on the x86. In *Proceedings of the USENIX Annual Technical Conference* (2008). To appear.

[15] GARFINKEL, T. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proceedings of Network and Distributed Systems Security Symposium (NDSS)* (2003), pp. 163–176.

[16] GARFINKEL, T., PFAFF, B., AND ROSENBLUM, M. Ostia: A delegating architecture for secure system call interposition. In *Proc. Network and Distributed Systems Security Symposium* (2004), pp. 187–201.

[17] GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. A. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of USENIX Security Symposium* (1996), pp. 1–13.

[18] Google Gears. http://gears.google.com/.

[19] HARREN, M., AND NECULA, G. C. Using dependent types to certify the safety of assembly code. In *Static Analysis Symposium (SAS)* (2005), pp. 155–170.

[20] JACKSON, C., BORTZ, A., BONEH, D., AND MITCHELL, J. Protecting browser state against Web privacy attacks. In *Proceedings of WWW* (2006).

[21] JOBS, S. Keynote address. Apple Worldwide Developers Conference, Aug. 2006.

[22] KICIMAN, E., AND LIVSHITS, B. AjaxScope: A platform for remotely monitoring the client-side behavior of Web 2.0 applications. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)* (2007), ACM.

[23] KOZUCH, M., AND SATYANARAYANAN, M. Internet suspend/resume. In *Proceedings of the Workshop on Mobile Computing Systems and Applications (WMCSA)* (2002), pp. 40–48.

[24] LIVSHITS, B., AND KICIMAN, E. Doloto: Code splitting for network-bound Web 2.0 applications. Tech. Rep. TR 2007-159, 2007.

[25] LORCH, J. R., AND SMITH, A. J. The VTrace tool: building a system tracer for Windows NT and Windows 2000. *MSDN Magazine 15*, 10 (2000), 86–102.

[26] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium* (2006), pp. 209–224.

[27] MCILROY, M. D. Mass produced software components. In *Software Engineering*, P. Naur and B. Randell, Eds. NATO Science Committee, Jan. 1969, pp. 138–150.

[28] MICROSOFT CORPORATION. Windows Driver Kit. http://microsoft.com/whdc/devtools/wdk/default.mspx.

[29] MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. From System F to typed assembly language. In *Symposium on Principles of Programming Languages (POPL)* (1998), pp. 85–97.

[30] NECULA, G. C., CONDIT, J., HARREN, M., MCPEAK, S., AND WEIMER, W. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst. 27*, 3 (2005), 477–526.

[31] NECULA, G. C., AND LEE, P. Safe kernel extensions without run-time checking. In *Proceedings of Operating Systems Design and Implementation (OSDI)* (1996), pp. 229–243.

[32] NIST Vulnerability Database. http://nvd.nist.gov/nvd.cfm.

[33] PRICE, D., AND TUCKER, A. Solaris zones: operating system support for server consolidation. In *Proceedings of the 18th Large Installation System Administration Conference (LISA)* (2004), pp. 241–254.

[34] PROVOS, N. Improving host security with system call policies. In *Proceedings of the 12th conference on USENIX Security Symposium (SSYM)* (2003), pp. 18–18.

[35] RINARD, M., CADAR, C., DUMITRAN, D., ROY, D., LEU, T., AND BEEBEE, J. Enhancing server

availability and security through failure-oblivious computing. In *Proceedings of Operating Systems Design and Implementation (OSDI)* (2004), pp. 303–316.

[36] ROBLES, G. Debian counting. http://libresoft.dat.escet.urjc.es/debian-counting/.

[37] SA7587. http://secunia.com/advisories/7587/.

[38] SABIN, T. Strace for NT. http://www.securityfocus.com/tools/1276.

[39] SAPUNTZAKIS, C. P., CHANDRA, R., PFAFF, B., CHOW, J., LAM, M. S., AND ROSENBLUM, M. Optimizing the migration of virtual computers. In *Proceedings of Operating Systems Design and Implementation (OSDI)* (2002), pp. 377–390.

[40] SourceForge. http://sourceforge.net.

[41] STOKELY, M., AND LEE, C. *The FreeBSD Handbook 3rd Edition, Vol. 1: User's Guide*. FreeBSD Mall, Inc., Brentwood, CA, 2003.

[42] SWIFT, M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. In *Proceedings of Symposium on Operating Systems Principles (SOSP)* (2003), pp. 207–222.

[43] TRIDGELL, A. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, 1999.

[44] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *Proceedings of Symposium on Operating Systems Principles (SOSP)* (1993), pp. 203–216.

[45] WALDSPURGER, C. A. Memory resource management in VMware ESX server. In *Proceedings of Operating Systems Design and Implementation (OSDI)* (2002), pp. 181–194.

[46] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Denali: Lightweight virtual machines for distributed and networked applications. Tech. Rep. 02-02-01, 2002.

[47] Wine Windows compatibility library. www.winehq.org.

[48] WITCHEL, E., RHEE, J., AND ASANOVIC, K. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *Proceedings of Symposium on Operating Systems Principles (SOSP)* (2005), pp. 31–44.

Notes

[1] The term "Rich Internet Application" or "RIA" is sometimes used to refer to these high-end apps. Since the distinction between a web app and an RIA is fuzzy at best, we consistently use the term "web app" herein.

[2] This value includes code supporting multiple platforms, and thus overestimates the size of the Xen TCB.

FlightPath: Obedience vs. Choice in Cooperative Services

Harry C. Li[1], Allen Clement[1], Mirco Marchetti[2], Manos Kapritsos[1], Luke Robison[1],
Lorenzo Alvisi[1], and Mike Dahlin[1]
[1]The University of Texas at Austin, [2]University of Modena and Reggio Emilia

Abstract: We present FlightPath, a novel peer-to-peer streaming application that provides a highly reliable data stream to a dynamic set of peers. We demonstrate that FlightPath reduces jitter compared to previous works by several orders of magnitude. Furthermore, FlightPath uses a number of run-time adaptations to maintain low jitter despite 10% of the population behaving maliciously and the remaining peers acting selfishly. At the core of FlightPath's success are *approximate equilibria*. These equilibria allow us to design incentives to limit selfish behavior rigorously, yet they provide sufficient flexibility to build practical systems. We show how to use an ε-Nash equilibrium, instead of a strict Nash, to engineer a live streaming system that uses bandwidth efficiently, absorbs flash crowds, adapts to sudden peer departures, handles churn, and tolerates malicious activity.

1 Introduction

We develop a novel approach to designing cooperative services. In a cooperative service, peers controlled by different entities work together to achieve a common goal, such as sharing files [13, 24] or streaming media [22, 26, 29]. Such a decentralized approach has several advantages over a traditional client-server one because peer-to-peer (p2p) systems can be highly robust, scalable, and adaptive. However, a p2p system may not see these benefits if it does not tolerate Byzantine peers that may disrupt the service or selfish peers that may use the service without contributing their fair share [3].

We propose *approximate equilibria* [11] as a rigorous and practical way to design cooperative services. Using these equilibria, we can design flexible mechanisms to tolerate Byzantine peers. More importantly, approximate equilibria guide how we design systems to incentivize selfish (or *rational*) peers to obey protocols.

Recent deployed systems [13, 24] and research prototypes [1, 3, 26, 29, 34] build incentives into their protocols because they recognize the need to curb rational deviations. These works fall into two broad categories.

The first set includes works that use incentives informally to argue that rational peers will obey a protocol. This approach provides system designers the freedom to engineer efficient and practical solutions. KaZaA [24] and BitTorrent [13] are examples of this approach. However, informally arguing correctness leaves systems open to subtle exploits in adversarial environments. For example, users can receive better service quality in the KaZaA network by running KaZaA Lite [25], a hacked binary that falsifies users' contributions. In a BitTorrent swarm, Sirivianos et al. [38] demonstrate how to freeride by connecting to many more peers than prescribed, thereby increasing the probability to be optimistically unchoked.

The second set of works emphasizes rigor by using game theory to design a protocol's incentives and punishments so that obeying the protocol is each rational peer's best strategy. This approach focuses on crafting a system to be a Nash equilibrium [35], in which no peer has an incentive to deviate unilaterally from its assigned strategy. The advantage of this more formal technique is that the resulting system is provably resilient to rational manipulation. The disadvantage is that strict equilibrium solutions limit the freedom to design practical solutions, yielding systems with several unattractive qualities. For example, BAR-Backup [3], BAR Gossip [29], and Equicast [26] do not allow dynamic membership, require nodes to waste network bandwidth by sending garbage data to balance bandwidth consumption, and provide little flexibility to adapt to changing system conditions.

The existing choices—practical but informal or rigorous but impractical—are discouraging, but approximate equilibria offer an alternative. These equilibria let us give a limited degree of choice to peers, departing from the common technique of eliminating choice to make a cooperative service a strict equilibrium.

In FlightPath specifically, approximate equilibria let us use run-time adaptations to tame the randomness of our gossip-based protocol, making it suitable for low jitter media streaming while retaining the robustness and load balancing of traditional gossip. The key techniques enabled by this flexibility include allowing a bounded imbalance between peers, redirecting load away from busy peers, avoiding trades with unhelpful peers, and arithmetic coding of data to provide more opportunities

for fruitful trades.

As a result of these dynamic adaptations, FlightPath is a highly efficient and robust media streaming service that has several attractive properties:

High quality streaming: FlightPath provides good service to every peer, not just good average service. In our experiments with over 500 peers, 98% of peers deliver every packet of an hour long video. 100% of peers miss less than 6 seconds.

Broad deployability: FlightPath uses a novel block selection algorithm to cap the peak upload bandwidth so that the protocol is accessible to users behind cable or ADSL connections.

Rational-tolerant: FlightPath is a $\frac{1}{10}$-Nash equilibrium under a reasonable cost model, meaning that rational peers have provably little incentive to deviate from the protocol. We define an ε-Nash equilibrium in Section 2.

Byzantine-tolerant: FlightPath provides good streaming quality despite 10% of peers acting maliciously to disrupt it.

Churn-resilient: FlightPath maintains good streaming quality while over 30% of the peer population may churn every minute. Further, it easily absorbs flash crowds and sudden massive peer departures.

Compared to our previous work [29], the above properties represent *both* a qualitative and quantitative improvement. We reduce jitter by several orders of magnitude and decrease the overhead of our protocol by 50% compared to BAR Gossip. Additionally, we allow peers to join and leave the system without disrupting service.

Although approximate equilibria provide weaker guarantees than strict ones, they can be achieved without relying on the strong assumptions needed by the existing systems that implement strict Nash equilibria. BAR Gossip assumes that rational participants only pursue short-sighted strategies, ignoring more sophisticated ones that might pay off in the long term. Equicast [26] assumes that a user is hurt by an infinite amount if it loses any packet of a stream. FlightPath does away with such assumptions, relying instead on the existence of a threshold below which few rational peers find it worthwhile to deviate.

We organize the rest of the paper as follows. Section 2 defines the live streaming problem and the model in which we are working. Section 3 describes FlightPath's basic trading protocol and discusses how to add flexibility to improve performance significantly and handle churn. We evaluate our prototype in Section 4 which looks at FlightPath without churn, with churn, and under attack. In Section 5, we analyze the incentives a rational peer may have to cheat. Finally, Section 6 highlights related work and Section 7 concludes this paper.

2 Problem & Model

We explore approximate equilibria in the context of streaming a live event over the Internet. A *tracker* maintains the current set of peers that subscribe to the live event. A *source* divides time into rounds that are r_{len} seconds long. In each round, the source generates num_ups unique stream packets that expire after $deadline$ rounds. The source multicasts each packet to a small fraction f of peers. All peers work together to disseminate those packets throughout the system. When a stream packet expires, all peers that possess that packet deliver it to their media application. If a peer delivers fewer than num_ups stream updates in a round, we consider that round *jittered* and our goal is to minimize such rounds. Our jitter metric is analogous to SecureStream's [22] continuity index—the ratio of packets delivered on time to total number of packets—when applied to rounds instead of just packets. We assume that the source and tracker nodes run as specified and do not fail, although we could relax this assumption using standard techniques for fault-tolerance [9, 39]. Peers, however, may fail.

We use the BAR model [3] to classify peer behaviors as Byzantine, altruistic, or rational. The premise of the BAR model is that when nodes can benefit by deviating, it may be untenable to bound the number of deviations to a small fraction. Thus, we desire to create protocols that continue to function even if all participants are rational and willing to deviate for a large enough gain.

While many nodes behave rationally, some may be Byzantine and behave arbitrarily because of a bug, misconfiguration, or ill-will. We assume that the fraction of nodes that are Byzantine is bounded by $F_{byz} < 1$. Altruistic peers obey the given protocol but may crash unexpectedly as can rational peers.

Non-Byzantine peers maintain clocks synchronized with the tracker. Nodes communicate over synchronous yet unreliable channels. We assume that each peer has exactly one public key bound to a permanent id. In practice, we can discharge this assumption by using a certificate authority or by implementing recent proposals to defend against Sybil attacks [16, 42].

We assume that cryptographic primitives—such as digital signatures, symmetric encryption, and one-way hashes—cannot be subverted. Our algorithms also require that private keys generate unique signatures [6]. We denote a message m signed by peer i as $\langle m \rangle_i$.

Finally, we hold peers accountable for the messages they send. We define a proof of misbehavior (POM) [3] as a signed message that proves a peer has deviated from the protocol. A POM against a peer is sufficient evidence for the source and tracker to evict a peer from the system, never letting that peer join a streaming session with that tracker or source in the future. We assume that eviction is a sufficient penalty to deter any rational peer from sending a message that the receiver could present as a POM.

2.1 Equilibrium Model

We analyze and evaluate FlightPath using ε-Nash equilibria [11]. In such an equilibrium, rational players deviate if and only if they expect to benefit by more than a factor of ε. This assumption is reasonable if switching protocols incurs a non-trivial cost such as effort to develop a new protocol, effort to install new software, or risk that new software will be buggy or malicious. Under such circumstances, it may not be worth the trouble to develop or use an alternate protocol. In FlightPath, we assume that protocols that bound the gain from cheating to $\varepsilon \leq \frac{1}{10}$ are sufficient to discourage rational deviations.

FlightPath is the first peer-to-peer system that is based on an approximate equilibrium. Other works [11, 14] have used approximate equilibria only when the strict versions have been computationally hard to calculate. To our knowledge, FlightPath is the first work to explore how these equilibria can be used to trade off resilience to rational manipulation against performance.

A peer's utility: We assume that a rational peer benefits from receiving a jitter-free stream, and that that benefit decreases as jitter increases. We also assume that a peer's cost increases proportionally with the upload bandwidth consumed. Although FlightPath is not tied to any specific utility function that combines these benefits and costs, we provide one here for concreteness: $u = (1-j)\beta - w\kappa$, where j is the average number of jitter events per minute, w is the average bandwidth used in kilobits per second, β is the benefit received from a jitter free data stream, and κ is the cost for each 1 kbps of upload bandwidth consumed. In Section 5, we show how the ratio of benefit to cost affects the ε we can bound in an ε-Nash equilibrium.

3 FlightPath Design

We discuss FlightPath's design in three iterations. In the first, we give an overview of a basic structure, inspired by BAR Gossip [29], that allows peers to trade updates with one another. We design trades to force rational peers to act faithfully in each trade until the last possible action, where deviating can save only negligible cost. This basic protocol allows few opportunities for a peer to game the system, but by the same token, it provides few options for dynamically adapting to phenomena like bad links, malicious peers, or overload. Therefore, in the second iteration, we describe how we add controlled amounts of choice to the basic trading protocol to improve its performance dramatically. In the third iteration, we show how to modify the protocol to deal with changing membership.

Readers familiar with related works on rational peers may be surprised to see that in the last two iterations we do not argue step-by-step about incentives. This difference is due to the flexibility of approximate equilibria, which allows optimizations that improve a user's start-to-finish benefits and costs, while still limiting any possible gains from cheating. In Section 5, we demonstrate that FlightPath is a $\frac{1}{10}$-Nash equilibrium under reasonable assumptions.

3.1 Basic Protocol

Prior to a live event, peers contact the *tracker* to join a streaming session. After authenticating each peer, the tracker assigns unique random member ids to peers and posts a static membership list for the session.

In each round, the *source* sends two kinds of updates: stream updates and linear digests. A *stream update* contains the actual contents of the stream. A *linear digest* [22] contains information that allows peers to check the authenticity of received stream updates. Linear digests are signed by the source and contain secure hashes of stream updates. We use linear digests in place of digitally signing every stream update to reduce the computational load and bandwidth necessary to run FlightPath. The source sends each of the *num_ups* unique stream updates for a round to a small fraction f of random peers in the system. When the source multicasts stream updates to selected peers at the beginning of every round, it also sends them the appropriate linear digests.

In each round, peers initiate and accept trades from their neighbors. As in BAR Gossip, a trade consists of four phases: partner selection, history exchange, update exchange, and key exchange. First, a peer selects a partner using a *verifiable pseudo-random algorithm* [29]. Second, partners exchange histories describing which updates they possess and which they still need. Partners use the histories to compute deterministically the exact updates they expect to receive and are obligated to send, under the constraint that partners exchange equal numbers of updates. Third, partners swap updates by encrypting them and sending the encrypted data in a briefcase message. Immediately afterwards, a peer sends a *promise* pledging that the contents of its briefcase

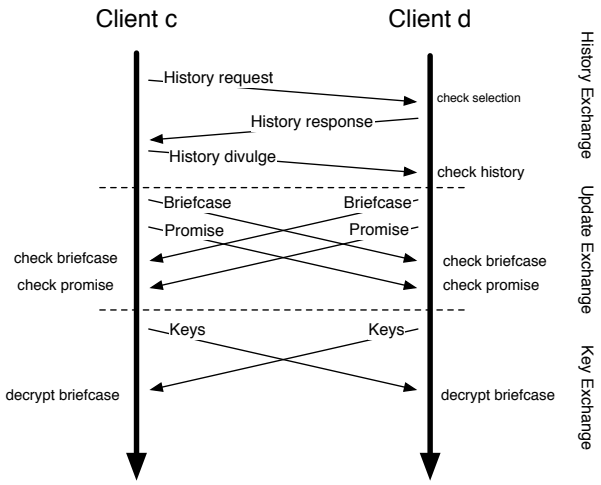


Figure 1: Illustration of a trade in the basic protocol.

is legitimate and not garbage data. Promises are the only digitally signed message in a trade; peers authenticate other messages using message authentication codes (MACs). Fourth, once a peer receives a briefcase and a matching promise message from its trading partner, that peer sends the decryption keys necessary to unlock the briefcase it sent.

These phases are similar to exchanges in BAR Gossip and they provide a similar guarantee: a rational peer has to upload the bulk of data in a trade to obtain any benefit from the trade. By deferring gratification and holding peers accountable via promise messages, we limit how much a cheating strategy can gain over obeying the protocol [29]. The main difference between a trade in this protocol compared to balanced exchanges in BAR Gossip is the addition of the promise.

We structure promises so that for each briefcase there is exactly one matching promise. Further, if a briefcase contains garbage data, then the matching promise is a proof of misbehavior (POM). Briefcases and promises provide this property because of how we intertwine these two kinds of messages. For each update u that a peer is obligated to send, that peer includes the pair $\langle u.id, u_{\langle \#u \rangle} \rangle$ in the briefcase it sends, where $u_{\langle \#u \rangle}$ denotes update u encrypted with a hash of itself. For each entry in the briefcase, the matching promise message contains a pair $\langle u.id, \#(u_{\langle \#u \rangle}) \rangle$. Therefore, if a briefcase holds garbage data, then the matching promise message would serve as a POM since that promise would contain at least one pair in which the hash for a self-encrypted update is wrong. Of course, a peer could upload garbage data in its briefcase but send a legitimate promise message to avoid sending a POM, but then the briefcase and promise would not match and that peer's partner would refuse to send the decryption keys.

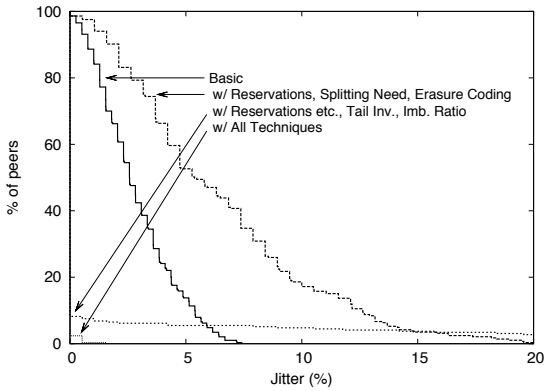


Figure 2: Reverse cumulative distribution of jitter.

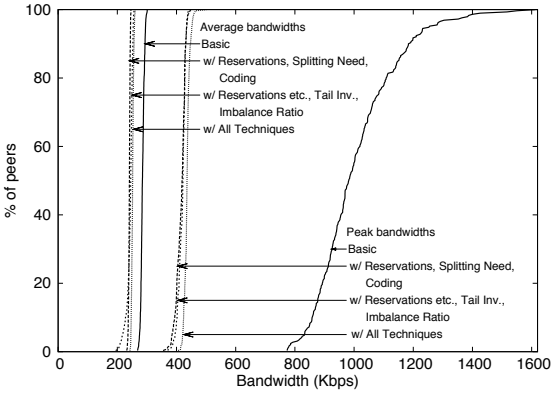


Figure 3: Cumulative distribution of average and peak bandwidths.

3.2 Taming Gossip

Gossip protocols are well-known for their robustness [7, 15] and are especially attractive in a BAR environment because their robustness helps tolerate Byzantine peers. However, while gossip's pair-wise interactions make crafting incentives easier than in a tree-based streaming system, it is reasonable to question whether that very randomness may make gossip inappropriate for streaming live data in which updates must be propagated to all nodes by a hard deadline.

In this section, we explain how the flexibility of approximate equilibria allows us to tame gossip's randomness by dynamically adapting run-time decisions. For concreteness, we show in Figures 2 and 3 how poorly the basic protocol performs when disseminating a 200 Kbps stream to 517 clients. In this experiment, the source generates $num\_ups = 50$ unique stream updates per round and sends each one to a random $f = 5\%$ of the peers. Updates expire $deadline = 10$ rounds from the time round in which they are sent. As the figure shows, the first three of the modifications we are about to discuss—

reservations, splitting need, and erasure coding—help in capping the peak bandwidth used by the protocol but, by reining in gossip's largesse with bandwidth, make jitter worse. The next three—tail inversion, imbalance ratio, and a trouble detector—reduce jitter by several orders of magnitude.

Reservations: One of the problems of using random gossip to stream live data is the widely variable number of trading partners a peer may have in any given round. In particular, although the expected number of trades in which a peer participates in each round is 2, the actual number varies widely, occasionally going past 8. Such high numbers of concurrent trades are undesirable for two reasons. First, a peer can be overwhelmed and be unable to finish all of its concurrent trades within a round. Figure 3 illustrates this problem as a high peak bandwidth in the basic protocol, making it impractical in bandwidth-constrained environments. Second, a peer is likely to waste bandwidth by trading for several duplicate updates when participating in many concurrent trades.

Rather than accept all incoming connections, FlightPath distributes the number of concurrent trades more evenly by providing a limited amount of flexibility in partner selection. The idea is simple. A peer c reserves a trade with a partner d before the round r in which that trade should happen. If d has already accepted a reservation for r, then c looks for a different partner. This straight-forward approach significantly reduces the probability of a peer committing to more than 2 concurrent trades in a round. At the same time, reservations also reduce the probability that a peer is only involved in the trade it initiates. The challenge in implementing reservations is how to give peers *verifiable* flexibility in their trading partners.

FlightPath provides each peer a small set of potential partners in each round. We craft this set carefully to address three requirements: peers need to select partners in a sufficiently random way to retain gossip's robustness, each peer needs enough choice to avoid overloaded or Byzantine peers, and these sets should be relatively unchanged if the population does not change much. Dynamic membership is discussed in Section 3.3, but its demands constrain the partner selection algorithm we describe here.

We force each peer to communicate with at least $\lfloor \log n \rfloor$ distinct neighbors by partitioning the membership list of n peers into $\lfloor \log n \rfloor$ bins and requiring a peer to choose a partner from a verifiable pseudorandomly chosen bin each round. Leitao et al. demonstrate that a set of gossip partners that grows logarithmically with system size can tolerate severe disruptions [28]. In

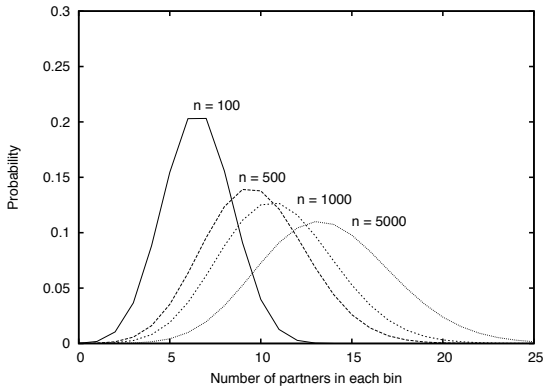


Figure 4: Distribution of view sizes in each bin for different membership list sizes. Graphs are calculated with $F_{byz} = 20\%$.

round r, peer c seeds a pseudo-random generator with $\langle r \rangle_c$, and uses the generator to select a bin; note that any peer can verify any other peer's bin selection.

Within a bin, we further restrict the nodes with whom a peer can communicate by giving each peer a *customized view* of each bin's membership based upon a peer's id. We define c's view to be all peers d such that the hash of c's member id with d's member id is less than some p. The tracker adjusts p according to inequality (1) so that almost every peer is expected to have at least one non-Byzantine partner in every bin. In the inequality, the expression $[1 - p(1 - F_{byz})]^{\frac{n}{\lceil \log n \rceil}}$ is the probability that for a given bin, a peer either has no partners or the partners it has are all Byzantine. Figure 4 gives an intuition for how this inequality affects a peer's choices as the system scales up.

$$[1 - [1 - p(1 - F_{byz})]^{\frac{n}{\lceil \log n \rceil}}]^{\lceil \log n \rceil} \geq 1 - \frac{1}{n} \qquad (1)$$

A peer c can use the choice provided by the combination of bins and views to reserve trades. A peer d that receives such a reservation verifies that c's view contains d and that $\langle r \rangle_d$ maps to the bin that contains d's entry in the membership list. If these checks pass, then d can either accept or reject the reservation.

As a general rule, peer d accepts a reservation only if it has not already accepted another reservation for the same round. Otherwise, d rejects the reservation, and c attempts a reservation with a different peer. Peer c can be exempt from this rule by setting a *plead* flag in its reservation, indicating that c has few options left. In this case, d accepts the reservation unless it has already committed to 4 trades in round r.

Splitting need: Reservations are effective in ensuring that peers are never involved in more than 4 concurrent trades. However, a peer that is involved in concurrent trades may still be overwhelmed with more data than it can handle during a round and may still receive too much duplicate data.

For example, consider a peer c involved in concurrent trades with peers d_0, d_1, d_2, and d_3. Peer c is missing 8 updates for a given round. The basic protocol may overwhelm c and waste bandwidth by having peers d_0–d_3 each send those 8 updates to c. Something more intelligent is for c's need to be split evenly across its trading partners, limiting each partner to send at most 2 updates. Note that while this scheme may be less wasteful than before, c now risks not receiving the 8 updates it needs since it is unlikely that its partners each independently select disjoint sets of 2 updates to exchange.

There seems to be a fine line between being conservative and receiving many duplicate updates to avoid jitter or taking a risk to save resources. We sidestep this tradeoff by using erasure coding [4, 30].

Erasure codes: Erasure coding has been used in prior works to improve content distribution [2, 12, 18, 27], but never to support live streaming in a setting with Byzantine participants. The source codes all of the stream data in a given round into $m > num\_ups$ stream updates such that any num_ups blocks are necessary and sufficient to reconstruct the original data. A peer stops requesting blocks for a given round once it has a sufficient number. Erasure coding reduces the probability that concurrent trades involve the same block.

In our experiments, we erasure code num_ups stream updates into $m = 2num\_ups$ blocks and modify the source to send each one to $\frac{f}{2}$ of the peers. In Figures 2 and 3, the source generates $2num\_ups = 100$ blocks and sends each one to a random 2.5% of the peers.

The modifications introduced so far reduce the protocol's peak bandwidth significantly, but at the cost of making jitter *worse*. We now describe three techniques that together nearly eliminate jitter without compromising the steps we have taken to keep the protocol from overwhelming any peer.

Tail inversion: As in many gossip protocols, the basic trading protocol biases recent updates over older ones to disseminate new data quickly. However, in a streaming setting, peers may sometimes value older updates over younger ones, for example when a set of older updates is about to expire and a peer desires to avoid jitter.

The drawback in preferring to trade for updates of an old round is that the received updates may not be useful in future exchanges because many peers may already possess enough updates to reconstruct the data streamed in that round. Indeed, an oldest-first bias performs very poorly in our prototype. Therefore, Flightpath provides a peer with the flexibility to balance recent updates that it can leverage in future exchanges against older updates that it may be missing. Instead of requesting updates in most-recent-first order, a peer has the option to receive updates from the two oldest rounds first and then updates in most-recent-first order. Alas, this particular ranking is not the fruit of deep insight—it is simply the one, out of the several we tried, that had the largest impact on reducing jitter: better rankings may well exist.

Imbalance ratio: The basic protocol balances trades so that a peer receives no more than it contributes in any round. Such equity can make it difficult for a peer that has fallen behind to recover.

FlightPath uses an *imbalance ratio* a to introduce flexibility into how much can be traded. Each peer tracks the number of updates sent to and received from its neighbors, ensuring that its credits and debits for each partner are within a of each other. We find that the imbalance ratio's most dramatic effect is that it allows individual trades to be very imbalanced if peers have longstanding relationships.

When a is set to 1, the trading protocol behaves like a traditional unbalanced gossip protocol, vulnerable to free-riding behavior [29]. When a is set to 0, every trade is balanced, offering little for rational peers to exploit, but also allowing unlucky peers to suffer significant jitter. We would like to set a to be as low as possible while maintaining low jitter; we found a = 10% to be a good tradeoff between these competing concerns.

Trouble Detector: Our final improvement takes advantage of the flexibility in selecting partners that our reservation mechanism offers. Each peer monitors its own performance by tracking how many updates it still needs for each round. If its performance falls below a threshold, then that peer can proactively initiate more than one trade in a round to avoid jitter. Peers treat this option as a safety net, as increasing the average number of concurrent trades also increases the average number of bytes uploaded to trade for each unique update.

We implement a simple detection module that informs a peer whether reserving more trades may be advisable. We assume that after each round a peer expects to double the number of updates that have not yet expired up to the point of possessing num_ups updates for each round. In practice, we find that peers typically gather updates more quickly than just doubling them. If a peer c notices that it possesses fewer updates than the

detection module advises, c schedules additional trades. Note that this is a local choice, based only on how many packets the peer has received for that round.

Figures 2 and 3 demonstrate the effectiveness of tail inversion, the imbalance ratio, and the trouble detector.

3.3 Flexibility for Churn

We now explain how to augment the protocol to handle churn. In FlightPath, the main challenge is in allowing peers to join an existing streaming session. Gossip's robustness to benign failures lends FlightPath a natural resilience to departures. However, the tracker still monitors peers to discover if any have left the system abruptly. Currently, we employ a simple pinging protocol, although we could use more sophisticated mechanisms as in Fireflies [40].

When a peer attempts to join a session, it expects to begin reliably receiving a stream without a long delay. As system designers, we have to balance that expectation against the resources available to get that peer up to speed. In particular, dealing with a flash crowd where the ratio of new peers to old ones is high presents a challenge. Moreover, in a BAR environment, we have to be careful in providing benefit to any peer who has not earned it. For example, if a single peer joins a system consisting of 50 peers, it may be desirable for all 50 to aid the new participant using balanced trades so that the new peer cannot free-ride off the system. However, consider the case when instead of 1 peer joining, 200 or 400 join. It is unreasonable to expect the original 50 to support a population of 400 peers who initially have nothing of value to contribute.

Below, we describe two mechanisms for allowing peers to join the system. The first allows the tracker to modify the membership list and to disseminate that list to all relevant peers. The second lets a new peer *immediately* begin trading so that it does not have to wait in silence until the tracker's list takes effect.

Epochs: A FlightPath tracker periodically updates the membership list to reflect joins and leaves. The tracker defines a new membership list at the beginning of each epoch, where the first epoch contains the first e_{len} rounds, the second epoch contains the next e_{len} rounds and so on. If a peer joins in epoch e, the tracker places that peer into the membership list that will be used in epoch $e+2$.

At the boundary between epochs e and $e+1$, the tracker shuffles the membership list for epoch $e+2$ and notifies the source of the shuffled list. Shuffling prevents Byzantine peers from attempting to position themselves at specific indices of the membership list, so as to take over a bin. Recall that we construct each peer's membership view to be independent of these indices so as not to end long-standing relationships prematurely.

After the tracker notifies the source of the next epoch's membership list, the source divides that list into pieces and places each piece into a third kind of update: *a partial membership list*. The source signs these lists and distributes them to peers as it would a stream update. Peers can trade partial membership lists just like they trade linear digests and stream updates. The only difference is that partial membership lists are given priority over all other updates in a trade and only expire when the epoch corresponding to that list ends. Once a peer obtains every partial membership list for an epoch, that peer can reconstruct the original membership list and use it to select trading partners.

Tub Algorithm: As described, a new peer would have to wait at least one epoch before it appears in the membership list and can begin to trade. FlightPath augments the static partner selection algorithm that uses bins with an online one that allows new peers to begin trading immediately without overwhelming the existing peers in the system. This algorithm also allows every peer to verify partner selections without global knowledge of how many peers joined nor of when they did so. Intuitively, our algorithm organizes all peers into *tubs* such that the first tub contains the peers in the current epoch's membership list and subsequent tubs contain peers who have recently joined. A peer selects partners from its own tub and also from any tub preceding its own. However, the probability that a peer from tub t selects from a tub $t' < t$ decreases geometrically with $t - t'$. This arrangement ensures that the load on a peer from all subsequent tubs is bound by a constant regardless of how many peers join. Figure 5 illustrates our algorithm.

For clarity, we describe our online algorithm assuming all peers have a global list that enumerates every peer in the system. Later, we show that this knowledge is unnecessary. The first n indices in this global list correspond to the n indices of the current epoch's membership list. The rest of the global list is sorted according to the order in which peers joined. We divide the global list into *tubs* where the first tub corresponds to the first n indices of the global list, the second tub to the next n indices, and so forth.

A peer c's membership view depends on its position in the global list. If c is in the first tub, its view and how it selects partners is unchanged from the static case (Section 3.2). If c is in a tub $t > 1$, c's view obeys three constraints:

1. Peer d is in c's view only if d precedes c in the list.

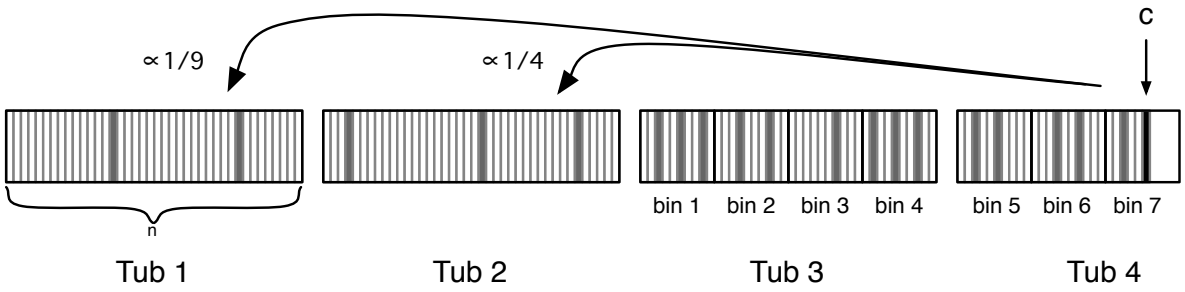


Figure 5: Illustration of the tub protocol from peer c's perspective. Shaded entries represent peers that c can contact for a trade when appropriate. Note that c only uses bins for its own tub and the immediately preceding one.

2. If d is in tub t or $t-1$, then d is in $c's$ view iff the hash of concatenating c's member id with d's member id is less than p (see inequality 1).
3. If d is in a tub $t' < t-1$, then d is in c's view iff the hash of concatenating c's member id and d's member id is less than a parameter p'.

The tracker adjusts p' according to inequality (2) so that almost every peer is expected to have at least one non-Byzantine partner in every tub.

$$[1-p'(1-F_{byz})]^n \leq \frac{1}{n} \tag{2}$$

A new peer c in tub $t_c > 1$ selects a trading partner for round r using two verifiable pseudo-random numbers, $rand_1$ and $rand_2$. First, c uses $rand_1$ to select a tub, exponentially weighting the selection towards its own tub. If c selects a tub $t < t_c - 1$, then c can trade with any peer in tub t that is also in c's view. If c selects either its own tub or the one immediately preceding its tub, then c uses $rand_2$ to make the final selection. c maps $rand_2$ to a bin starting from the first bin in tub $t-1$ and ending with c's own bin. From the selected bin, c can trade with any peer in its view.

If every peer knew the global list, then it would be straight-forward to select and verify trading partners. Fortunately, this global knowledge is unnecessary: to select trading partners, a newly joined peer only needs to know the peers in its own view, the epoch in which those peers joined the system, and the indices of those peers in the global list. When a peer c joins the system, c obtains such information directly from the tracker.

To verify that a peer c selects a partner d appropriately, d needs to know c's index in the global membership list. The tracker encodes such information in a *join token* that it gives to c when c joins the system. The join token specifies c's index in the global list for the two epochs until c is part of an epoch's membership list. c includes its join token in its reservation message to d.

4 Evaluation

We now show that FlightPath is a robust p2p live streaming protocol. Through experiments on over 500 peers, we demonstrate that FlightPath:

- Reduces jitter by several orders of magnitude compared to BAR Gossip
- Caps peak bandwidth usage to within the constraints of a cable or ADSL connection
- Maintains low jitter and efficiently uses bandwidth despite flash crowds
- Recovers quickly from sudden peer departures
- Continues to deliver a steady stream despite churn
- Tolerates up to 10% of peers acting maliciously

4.1 Methodology

We use FlightPath to disseminate a 200 Kbps data stream to several hundred peers distributed across Utah's Emulab and UT Austin's public Linux machines. In most experiments, we use 517 peers, but drop to 443 peers in the churn and Byzantine experiments as the availability of Emulab machines declined. We run each experiment 3 times. When we present cumulative distributions, we combine points from all three experiments. We include standard deviation when doing so keeps figures readable.

In our experiments, rounds last 2 seconds and epochs last 40 rounds. In each round, the source sends 100 Reed-Solomon coded stream updates and 2 linear digests. 50 stream updates are necessary and sufficient to reconstruct the original data. Stream updates expire 10 rounds after they are sent. The source sends each stream update to a random 2.5% of peers. Stream updates are 1072 bytes long, while linear digests are 1153 bytes long.

We implement FlightPath in Python using MD5 for secure hashes and RSA-FDH with 512 bit keys for digital signatures. Peers exchange public certificates and

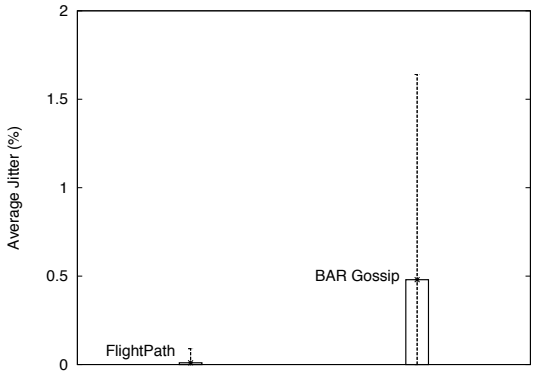


Figure 6: Average jitter in FlightPath and BAR Gossip peers. (n = 517)

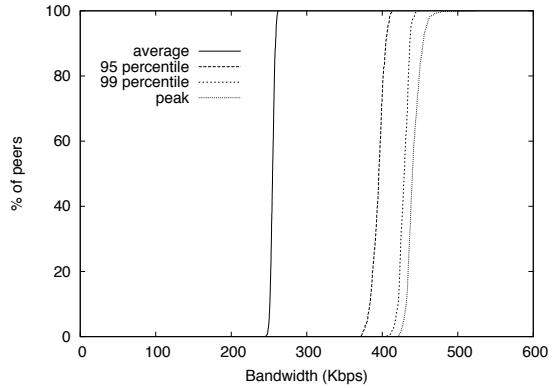


Figure 7: Distributions of peers' average, 95 percentile, 99 percentile, and peak bandwidths. (n = 517)

agree on secret keys for MACs a few seconds before reserving trades with one another. Peers also set the budget for how many updates they are willing to upload in a round to $\mu = 100$, which is split evenly across concurrent trades.

Steady State Operation: In the first experiment, we run FlightPath on 517 peers to assess its performance under a relatively well-behaved and static environment. Figure 6 shows that the average jitter of FlightPath is orders of magnitude lower than BAR Gossip. Of the three experiments we ran for one hour, the worst jitter was in an experiment in which 1 peer missed 6 seconds of video, 5 peers missed 4 seconds, and 3 peers missed 2 seconds. All jitter events occurred during the first minute. Figure 7 confirms that peers use approximately 250 Kbps on average and also depicts cumulative distributions tracing the peak bandwidth of each peer along with curves for the 99 and 95 percentile bandwidth curves. As in Section 3.2, the combination of reservations, splitting a peer's need and erasure coding is effective in capping peak bandwidth.

Joins: We now evaluate how well FlightPath handles joins into the system. In particular, we stress the tub algorithm, described in Section 3.3, to handle large populations of peers who seek to join a streaming session all at once. In this experiment, we start a session with 50 peers. In round 40, varying numbers of peers simultaneously attempt to join the system. As Figure 8 illustrates, the average bandwidth of the original peers noticeably spikes immediately after round 40 and settles to a higher level than before. In round 120, when new peers are integrated into the membership list, average bandwidth of the original 50 drops back to its previous levels. As shown, FlightPath peers are relatively unaffected by joining events. None of the original 50

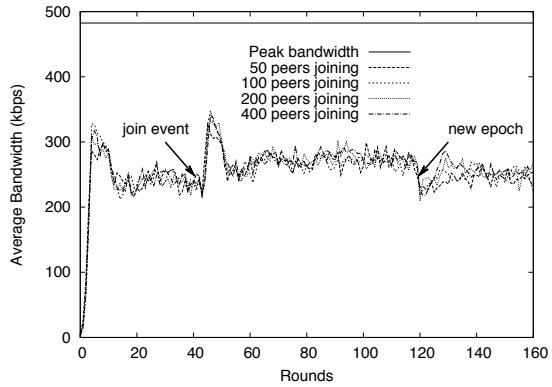


Figure 8: Bandwidth of peers already in the system with different sized flash crowds. (n = 50)

peers experienced a jitter event during any of these experiments. Also note that the peak bandwidth across all three runs of each experiment was 482.5 Kbps.

Figure 9 depicts the number of rounds a peer may have to wait before it begins to deliver a stream reliably. We define the round in which a peer reliably begins to deliver a stream as the first round in which a peer experiences no jitter for three rounds. Interestingly, we see that if more peers join, performance improves. This effect can be explained by our tub algorithm. The peers in the last tub are contacted the least. In the experiment in which only 50 peers join, all of the newly joined peers are in the last tub. The last tub in the experiment with 400 peers joining has a similar problem, but the last tub is masked by the success of the preceding 7 tubs.

Departures: Figure 10 shows FlightPath's resilience to large fractions of a population suddenly departing. Departing peers exit abruptly without notifying the tracker or completing reserved trades. The figure shows the percentage of peers jittered after a massive departure event

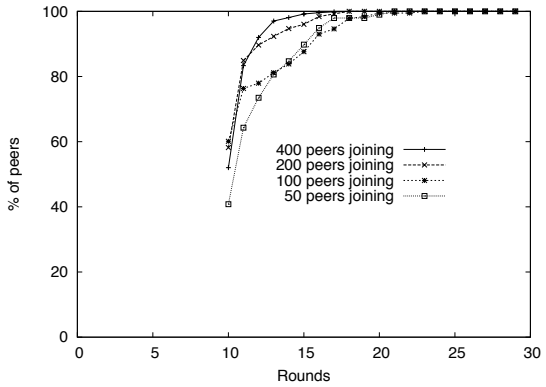


Figure 9: CDF of join delays for different size joining crowds. ($n = 50$)

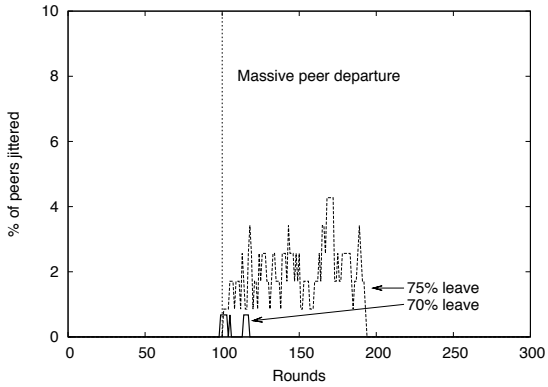


Figure 10: Jitter during massive departure. ($n = 517$)

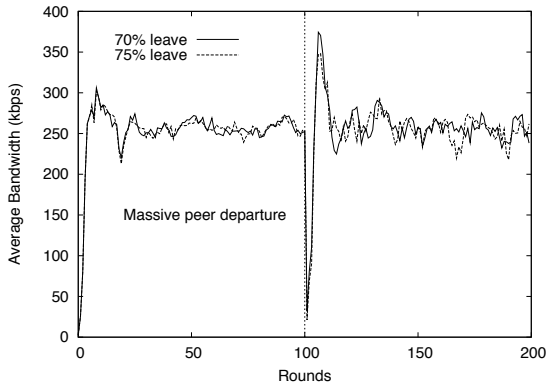


Figure 11: Average bandwidth after a massive departure.($n = 517$)

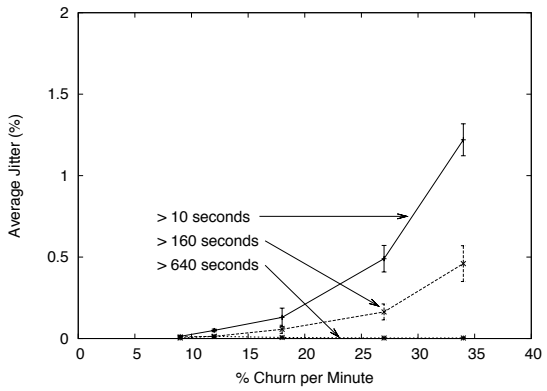


Figure 12: Average jitter as churn increases.($n = 443$)

of 70% and 75% of random peers. We chose these fractions because smaller fractions had little observable effect with respect to jitter. The figure shows that there exists a threshold between 70% and 75% in which FlightPath cannot tolerate any more departures.

FlightPath's resilience to such massive departures is a consequence of a few traits. First, peers discover very quickly whether potential partners have left or not via the reservation system. Second, peers have choice in their partner selection, so they can avoid recently departed peers. Finally, each peer's trouble detector helps in reacting quickly to avoid jitter. Figure 11 shows the effect of the trouble predictor. Average bandwidth of remaining peers drops dramatically after the leave event, but then spikes sharply to make up for missed trading opportunities.

Churn: We now evaluate how FlightPath performs under varying amounts of churn. In our experiments, peers join and then leave after an exponentially random amount of time. Because short-lived participants are proportionally more affected by their start-up transients, our presentation segregates peers by the amount of time they remain in the system. Figure 12 shows average jitter as we increase churn. The average jitter of peers who join the system for at least 10 seconds steadily increases with churn. Peers who stay in the system for at least 640 seconds experience very little jitter even when 37% of peers churn every minute. Further experiments (not included) show that there is a non-negligible probability of being jittered during the first two minutes after joining a streaming session. Afterwards, the chance of being jittered falls to nearly 0.

Figure 13 shows that churn does manifest as increasing join delays for new peers. We see that the time needed to join a session is unacceptable under high amounts of churn. This quality points to a weakness of FlightPath and suggests a need for a bootstrapping mechanism for new peers. However, care needs to be exercised in not allowing peers to game the system by abusing the bootstrapping mechanism to obtain updates without uploading.

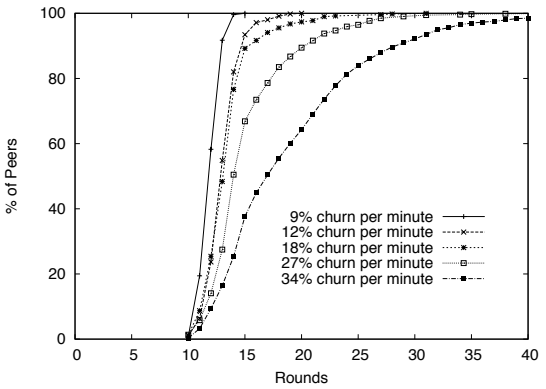


Figure 13: Join delay under churn.(n = 443)

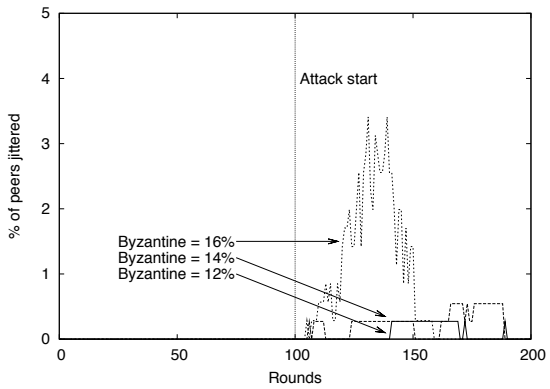


Figure 14: Jitter with malicious peers. (n = 443)

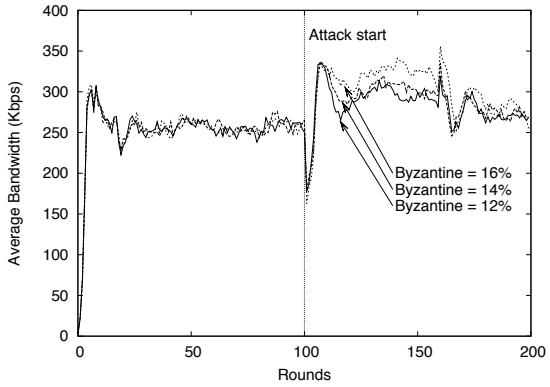


Figure 15: Bandwidth with malicious peers. (n = 443)

Malicious attack: In this experiment, we evaluate FlightPath's ability to deliver a stream reliably in the presence of Byzantine peers. While any peer whose utility function is unknown is strictly speaking Byzantine in our model, we are especially interested in understanding how FlightPath behaves under attack, when Byzantine peers behave maliciously.

Although Byzantine peers cannot make a non-Byzantine peer deliver an inauthentic update, they can harm the system by degrading its performance. We have studied the effect on jitter of several malicious strategies—we report here the results for the one that appeared to cause the greatest harm. According to this scheme, malicious peers act normally for the first 100 rounds of the protocol. However, starting in round 100, they initiate as many trades as they can and respond positively to all trade reservations, seeking to monopolize as many trades in the system as possible. The Byzantine peers participate in the history exchange phase of a trade but in no subsequent phase. In a history exchange, a Byzantine peer reports that it has all the updates that are less than 3 rounds old and is missing all the other updates. This strategy commits a large amount of its partners bandwidth to the exchange. Ultimately, however, non-Byzantine peers find trades with Byzantine ones useless.

Figure 14 shows the percentage of peers jittered when 12%, 14%, and 16% of peers behave in this malicious way. We elide the experiment in which 10% of peers are Byzantine because no peer suffered jitter in those experiments. Figure 15, which depicts the average bandwidth of non-Byzantine peers, is similar to the one in which peers abruptly leave the system. The subtle difference is that the average bandwidth used remains higher with more Byzantine peers.

Wide Area Network: Finally, we evaluate how FlightPath performs under wide area network conditions. In this experiment, we use 300 clients on a local area network but delay all packets between clients according to measured Internet latencies. We assign each client a random identity from the 1700+ hosts listed in the King data set of Internet latencies [19]. We use the data set to delay every packet according to its source and destination.

As in the case without added delays, all jitter events occurred in the first minute of the experiments. Figure 16 depicts the average percentage of peers jittered in the first minute, the average upload bandwidth, and the peak upload bandwidth for our experiments with the added delays and without. Aside from a slight increase (almost 10 Kbps) in average upload bandwidth, peak upload bandwidth rose by approximately 40 Kbps. These increases are the result of some exchanges not completing by the end of a round, requiring peers involved to make up for the loss in subsequent rounds.

5 Equilibria Analysis

In contrast to previous rigorous approaches to dissuade rational deviation, FlightPath does not ensure that every step of the protocol is in every peer's best interest.

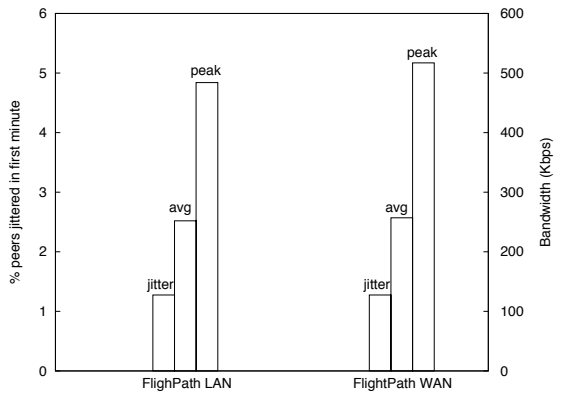


Figure 16: Bandwidth under WAN conditions. ($n = 300$)

Indeed, it is easy to imagine circumstances in which a peer might benefit from deviating, for example, by setting the *plead* flag early to increase the likelihood that a selected peer will accept its invitation. Instead, FlightPath ensures an ε-Nash equilibrium in which no peer can significantly improve its overall utility regardless of how it makes these individual choices.

The high level argument is simple. A peer can only increase its utility by obtaining more benefit (receiving less jitter) or reducing cost (uploading fewer bytes). Since we engineered FlightPath to provide very low jitter in a wide range of environments, a peer has very little ability to obtain more benefit. With respect to decreasing costs, we structure trades so that a rational peer has to pay at least $\lceil \frac{1}{1+a} \rceil$ of the cost of uploading x updates in order to receive x updates, where a is the imbalance ratio.

We now develop this argument more formally to bound the added utility that can be gained by a peer that deviates. We analyze FlightPath in the steady state case and ignore transient start-up effects or end game scenarios, which would matter little in the overall utility of watching something as long as a movie.

We begin by revisiting the utility function $u = (1 - j)\beta - w\kappa$. Recall that j is the average number of jitter events per minute, β is the benefit from watching a jitter-free stream, w is the average upload bandwidth used in Kbps, and κ is the cost per Kbps. If we let the expected utility of an optimal cheating strategy be $u_o = (1 - j_o)\beta - w_o\kappa$ and the expected utility of obeying the protocol be $u_e = (1 - j_e)\beta - w_e\kappa$, then we can express ε as follows:

$$\varepsilon = \frac{u_o - u_e}{u_e} = \frac{(j_e - j_o)\beta - (w_o - w_e)\kappa}{(1 - j_e)\beta - w_e\kappa} \qquad (3)$$

We simplify equation 3 with the following assumptions: *i)* the benefit of running FlightPath exceeds the cost, *ii)* the optimal cheating strategy receives no jitter, and *iii)* the optimal cheating strategy uses a fraction $b < 1$ of the bandwidth of running the protocol. These assumptions let us express ε as a function of the benefit-to-cost ratio c, the expected number of jitter events per minute j_e, and the proportional savings in cost $1 - b$.

Parameter	Description
num_ups	num stream updates per round needed
m	num stream updates per round
f	fraction of updates received from source
$budget$	max num of updates sent in a round
a	imbalance ratio

Table 1: Summary of the analysis parameters.

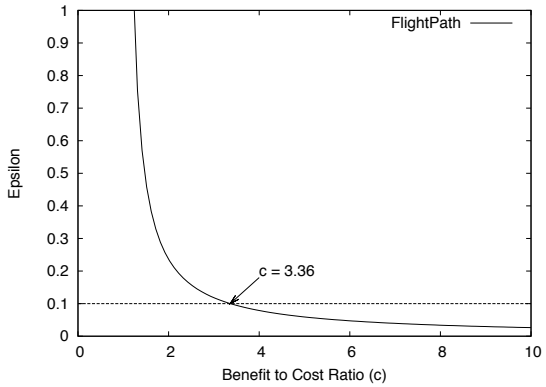


Figure 17: ε as a function of the benefit to cost ratio.

$$\varepsilon = \frac{\frac{c j_e}{1 - j_e} + (1 - b)}{c - 1} \qquad (4)$$

As the jitter expected is an empirical phenomenon, we use our evaluation to determine j_e, which after the first minute is 0. We then establish a lower bound on b using parameters specific to our system, listed in Table 1.

In the steady state, a peer p following a hypothetical optimal strategy participates on average in at least one trade every $t = \lfloor \frac{budget}{num\_ups - mf} \rfloor$ rounds. Furthermore, the average number of updates that it needs in each trade is $needed = t(num\_ups - mf)$. Assuming that p is lucky or clever enough to upload no more updates than it has to in all trades, then p still uploads at least $min\_up = \lceil \frac{needed}{1+a} \rceil$ updates on average in each trade.

Let γ be the fixed cost in kilobits of a trade and let ρ be the increase in cost of a trade for each update p uploads. Then the average cost that p has to pay for each trade is $\gamma + min\_up \times \rho$. Given the message encodings in our prototype, the fixed cost of a trade is 305 bytes and the increase for each uploaded update is 1104 bytes. These

values correspond to $\gamma = 2.44$ and $\rho = 8.832$. Our goal is to ensure that the net utility of the optimal strategy is not significantly more than for FlightPath's strategy. For $\varepsilon = \frac{1}{10}$, solving for c in Equation 3 indicates that FlightPath is a $\frac{1}{10}$-Nash equilibrium as long as the user values the stream at least 3.36 times as much as the bits uploaded to participate in the system. Figure 17 illustrates the ε value FlightPath provides for each benefit-to-cost ratio.

6 Related Work

This work builds on a broad set of approaches for content dissemination and Byzantine or rational-tolerant protocols.

Clearly, BAR Gossip [29] is the work most closely related to FlightPath. We explain how it is similar and different from FlightPath throughout this paper, and in particular in Section 3.

Several tree-based overlays [10, 23] have been devised to disseminate streaming data. Ngan et al. [36] suggest that restructuring Splitstream [10] trees can guard against free-riders by periodically changing the parent-child relationships among peers, a communication pattern that begins to resemble gossip. Chunkyspread [41] uses a multi-tree based approach to multicast. Chunkyspread builds random trees using low overhead bloom filters and allows peers to make local decisions to tune the graph for better performance.

In Araneola [33], Melamed and Keidar construct random overlay graphs to multicast data. They show that Araneola's overlay structure achieves mathematical properties important for low-latency, load balancing, and resilience to benign failures.

Demers et al. introduced gossip protocols to manage consistency in Xerox's Clearinghouse servers [15]. Years later, Birman et al. [7] used gossip to build a probabilistic multicast—a middle ground between existing reliable multicast and best effort multicast protocols. Since then, many have explored ways to improve gossip's throughput and robustness [8, 17, 20, 21, 28].

None of the above works consider Byzantine peers who can harm the system by spreading false messages. One can guard against such attacks by using techniques that avoid digital signatures [31, 32], but signatures can dramatically simplify protocols and are used in many practical gossip implementations [8, 22, 29, 40].

Haridasan and van Renesse [22] build a Byzantine fault-tolerant live streaming system over the Fireflies system. Their system, SecureStream, introduces *linear digests* to efficiently authenticate stream packets. As in CoolStreaming [43] and Chainsaw [37], SecureStream also uses a pull-based gossip protocol to reduce the number of redundant sends.

Badishi et al. [5] show in DRUM how gossip protocols can resist Denial-of-Service (DoS) attacks by resource bounding public ports and port hopping. We could integrate DRUM's techniques into FlightPath.

To our knowledge, Equicast [26] is the first work to address formally rational behavior in multicast protocols. Equicast organizes peers into a random graph over which it disseminates content. The authors prove Equicast is an equilibrium, but assume that rational peers lack the expertise to modify the protocol beyond tuning the cooperation level. Currently, Equicast is a purely theoretical work, making an empirical comparison with FlightPath difficult.

BAR-Backup [3] is a p2p backup system for Byzantine and rational peers. Peers implement a replicated state machine that moderates interactions between peers to ensure that peers behave appropriately.

7 Conclusion

We present approximate equilibria as a new way to design cooperative services. We show that approximate equilibria allow us to provably limit how much selfish participants can gain by deviating from a protocol. At the same time, these equilibria provide enough freedom to engineer practical solutions that are flexible enough to handle many adverse situations, such as churn and Byzantine peers.

We use ε-Nash equilibria, an example of an approximate equilibrium, to design FlightPath, a novel p2p live streaming system. FlightPath improves on the existing state-of-the-art both qualitatively and quantitatively, reducing jitter by several orders of magnitude, using bandwidth efficiently, handling churn, and adapting to attacks. More broadly, FlightPath demonstrates that we do not have to sacrifice rigor to engineer Byzantine and rational-tolerant systems that perform well and operate efficiently.

8 Acknowledgements

The authors would like to thank the anonymous reviewers and our shepherd, Dejan Kostić. Special thanks to Petros Maniatis and Taylor Riché for their comments on earlier versions of this work. This project was supported in part by NSF grant CSR-PDOS 0509338.

References

[1] I. Abraham, D. Dolev, R. Gonen, and J. Halpern. Distributed computing meets game theory: robust mechanisms for rational secret sharing and multiparty computation. In *Proc. 25th PODC*, pages 53–62, July 2006.

[2] R. Ahlswede, N. Cai, S.-Y. Li, and R. Yeung. Network information flow. *Information Theory, IEEE Transactions on*, 46(4):1204–1216, Jul 2000.

[3] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proc. 20th SOSP*, pages 45–58, Oct. 2005.

[4] N. Alon, J. Edmonds, and M. Luby. Linear time erasure codes with nearly optimal recovery. In *FOCS '95*, page 512, Washington, DC, USA, 1995. IEEE Computer Society.

[5] G. Badishi, I. Keidar, and A. Sasson. Exposing and eliminating vulnerabilities to denial of service attacks in secure gossip-based multicast. In *Proc. DSN-2004*, page 223, Washington, DC, USA, 2004. IEEE Computer Society.

[6] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *Proc. 1st CCC*, pages 62–73, New York, NY, USA, 1993. ACM Press.

[7] K. P. Birman, M. Hayden, O. Oskasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM TOCS*, 17(2):41–88, May 1999.

[8] K. P. Birman, R. van Renesse, and W. Vogels. Spinglass: Secure and scalable communications tools for mission-critical computing. In *DARPA DISCEX-2001*, 2001.

[9] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM TOCS*, 14(1):80–107, 1996.

[10] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: high-bandwidth multicast in cooperative environments. In *Proc. 19th SOSP*, pages 298–313. ACM Press, 2003.

[11] S. Chien and A. Sinclair. Convergence to approximate nash equilibria in congestion games. In *SODA '07*, pages 169–178, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[12] P. A. Chou, Y. Wu, and K. Jain. Practical network coding. In *ACCCC03*, October 2003.

[13] B. Cohen. Incentives build robustness in BitTorrent. In *P2PECON '03*, June 2003.

[14] C. Daskalakis, A. Mehta, and C. Papadimitriou. A note on approximate nash equilibria. In *WINE '06*, 2006.

[15] A. Demers, D. Greene, C. Houser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. 11th SOSP*, Aug. 1987.

[16] J. R. Douceur. The Sybil attack. In *Proc. 1st IPTPS*, pages 251–260. Springer-Verlag, 2002.

[17] P. Eugster, S. Handurukande, R. Guerraoui, A. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *DSN '01*, pages 254–269, July 2001.

[18] C. Gkantsidis and P. Rodriguez. Network coding for large scale content distribution. *INFOCOM*, 4:2235–2245 vol. 4, March 2005.

[19] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: estimating latency between arbitrary internet end hosts. *SIGCOMM Comput. Commun. Rev.*, 32(3):11–11, 2002.

[20] I. Gupta, K. Birman, and R. van Renesse. Fighting fire with fire: using randomized gossip to combat stochastic scalability limits. *Journal of Quality and Reliability Engineering International*, 18(3):165–184, 2002.

[21] I. Gupta, A.-M. Kermarrec, and A. J. Ganesh. Efficient and adaptive epidemic-style protocols for reliable and scalable multicast. *IEEE TPDS*, 17(7):593–605, 2006.

[22] M. Haridasan and R. van Renesse. Defense against intrusion in a live streaming multicast system. In *Proceedings of P2P '06*, pages 185–192, Washington, DC, USA, 2006. IEEE Computer Society.

[23] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. James W. O'Toole. Overcast: reliable multicasting with an overlay network. In *Proceedings of OSDI '00*, pages 14–14, Berkeley, CA, USA, 2000. USENIX Association.

[24] Kazaa. http://www.kazaa.com.

[25] Kazaa Lite. http://en.wikipedia.org/wiki/Kazaa_Lite.

[26] I. Keidar, R. Melamed, and A. Orda. Equicast: Scalable multicast with selfish users. In *PODC '06*, 2006.

[27] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In *SOSP '03*, pages 282–297, New York, NY, USA, 2003. ACM.

[28] J. Leitao, J. Pereira, and L. Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *DSN '07*, pages 419–429, Washington, DC, USA, 2007. IEEE Computer Society.

[29] H. C. Li, A. Clement, E. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. BAR Gossip. In *Proceedings of OSDI '06*, pages 191–204, Nov. 2006.

[30] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, D. A. Spielman, and V. Stemann. Practical loss-resilient codes. In *STOC '97*, pages 150–159. ACM Press, 1997.

[31] D. Malkhi, Y. Mansour, and M. K. Reiter. Diffusion without false rumors: on propagating updates in a byzantine environment. *TCS*, 299(1-3):289–306, 2003.

[32] D. Malkhi, M. Reiter, O. Rodeh, and Y. Sella. Efficient update diffusion in Byzantine environments. In *Proc. 20th SRDS*, 2001.

[33] R. Melamed and I. Keidar. Araneola: A scalable reliable multicast system for dynamic environments. In *Proc. of NCA '04*, pages 5–14, Washington, DC, USA, 2004. IEEE Computer Society.

[34] T. Moscibroda, S. Schmid, and R. Wattenhofer. When selfish meets evil: Byzantine players in a virus inoculation game. In *Proc. 25th PODC*, pages 35–44, July 2006.

[35] J. Nash. Non-cooperative games. *The Annals of Mathematics*, 54:286–295, Sept 1951.

[36] T.-W. Ngan, D. S. Wallach, and P. Druschel. Incentives-compatible peer-to-peer multicast. In *P2PECON '04*, 2004.

[37] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. Mohr. Chainsaw: Eliminating trees from overlay multicast. In *IPTPS '05*, February 2005.

[38] M. Sirivianos, J. H. Park, R. Chen, and X. Yang. Free-riding in bittorrent networks with the large view exploit. In *IPTPS '07*, February 2007.

[39] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Comm. ACM*, 39(4):76–83, 1996.

[40] R. van Renesse, H. Johansen, and A. Allavena. Fireflies: Scalable support for intrusion-tolerant overlay networks. In *EuroSys '06*, 2006.

[41] J. Venkataraman, P. Francis, and J. Calandrino. Chunkyspread: Multi-tree unstructured peer-to-peer multicast. In *IPTPS '06*, February 2006.

[42] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. Sybilguard: Defending against sybil attacks via social networks. In *ACM SIGCOMM '06*, Sept.

[43] X. Zhang, J. Liu, B. Li, and T. P. Yum. CoolStreaming/DONet: A data-driven overlay network for live media streaming. In *IEEE INFOCOM*, Mar. 2005.

Mencius: Building Efficient Replicated State Machines for WANs

Yanhua Mao
CSE, UC San Diego
San Diego, CA - USA
maoyanhua@cs.ucsd.edu

Flavio P. Junqueira
Yahoo! Research Barcelona
Barcelona, Catalonia - Spain
fpj@yahoo-inc.com

Keith Marzullo
CSE, UC San Diego
San Diego, CA - USA
marzullo@cs.ucsd.edu

Abstract

We present a protocol for general state machine replication – a method that provides strong consistency – that has high performance in a wide-area network. In particular, our protocol *Mencius* has high throughput under high client load and low latency under low client load even under changing wide-area network environment and client load. We develop our protocol as a derivation from the well-known protocol Paxos. Such a development can be changed or further refined to take advantage of specific network or application requirements.

1 Introduction

The most general approach for providing a highly available service is to use a replicated state machine architecture [33]. Assuming a deterministic service, the state and function is replicated across a set of servers, and an unbounded sequence of consensus instances is used to agree upon the commands they execute. This approach provides strong consistency guarantees, and so is broadly applicable. Advances in efficient consensus protocols have made this approach practical as well for a wide set of system architectures, from its original application of embedded systems [34] to asynchronous systems. Recent examples of services that use replicated state machines include Chubby [6, 8], ZooKeeper [37] and Boxwood [28].

With the rapid growth of wide-area services such as web services, grid services, and service-oriented architectures, a basic research question is how to provide efficient state machine replication in the wide area. One could choose an application – for example, atomic commit in a service-oriented architecture, and provide an efficient solution for that application (for a large client base and high throughput). Instead, we seek a general solution that only assumes the servers and the clients are spread across a wide-area network. We seek high performance: both high throughput under high client load and low latency under low client load in the face of changing wide-area network environment and client load. And, we seek a solution that comes with a derivation, like the popular consensus protocol Paxos has [22], so it can be modified to apply it to a specific application [15].

Existing protocols such as Paxos, Fast Paxos [25], and CoReFP [13] are not, in general, the best consensus protocols for wide-area applications. For example, Paxos relies on a single leader to choose the request sequence. Due to its simplicity it has high throughput, and requests generated by clients in the same site as the leader enjoy low latency, but clients in other sites have higher latency. In addition, the leader in Paxos is a bottleneck that limits throughput. Having a single leader also leads to an unbalanced communication pattern that limits the utilization of bandwidth available in all of the network links connecting the servers. Fast Paxos and CoReFP, on the other hand do not rely on a single leader. They have low latency under low load, but have lower throughput under high load due to their higher message complexity.

This paper presents *Mencius*[1], a multi-leader state machine replication protocol that derives from Paxos. It is designed to achieve high throughput under high client load and low latency under low client load, and to adapt to changing network and client environments.

The basic approach of Mencius is to partition the sequence of consensus protocol instances among the servers. For example, in a system with three servers, one could assign to server 0 the consensus instances 0, 3, 6 etc, server 1 the consensus instances 1, 4, 7, etc and server 2 the consensus instances 2, 5, 8 etc. Doing this amortizes the load of being a leader, which increases throughput when the system is CPU-bound. When the network is the bottleneck, a partitioned leader scheme more fully utilizes the available bandwidth to increase throughput. It also reduces latency, because clients can use a local server as the leader for their requests; because of the design of Mencius, a client will typically not have to wait for its server to get its turn.

The idea of partitioning sequence numbers among

multiple leaders is not original: indeed, it is at the core of a recent patent [26], for the purpose of amortizing server load. To the best of our knowledge, however, Mencius is novel: not only are sequence numbers partitioned, key performance problems such as adapting to changing client load and to asymmetric network bandwidth are addressed. Mencius accomplishes this by building on a simplified version of consensus that we call *simple consensus*. Simple consensus allows servers with low client load to skip their turns without having to have a majority of the servers agree on it first. By opportunistically piggybacking SKIP messages on other messages, Mencius allows servers to skip turns with little or no communication and computation overhead. This allows Mencius to adapt inexpensively to client and network load variance.

The remainder of the paper is as follows. Section 2 describes the wide-area system architecture for which Mencius is designed. Paxos and its performance problems under our system architecture is described in Section 3. Section 4 refines Paxos into Mencius. Section 5 discusses a flexible commit mechanism that reduces latency. Section 6 discusses how to choose parameters. Section 7 evaluates Mencius, Section 8 summarizes related work, and Section 9 discusses future work and open issues. Section 10 concludes the paper.

2 Wide-area replicated state machines

We model a system as n sites interconnected by a wide-area network. Each site has a server and a group of clients. These run on separate processors and communicate through a local-area network. The wide-area network has higher latency and less bandwidth than the local-area networks, and the latency can have high variance. We model the wide-area network as a set of links pairwise connecting the servers. The bandwidth between pairs of servers can be asymmetric and variable.

We do not explicitly consider any dependent behavior of these links. For example, we do not consider issues such as routers that are bottlenecks for communication among three or more sites. This assumption holds when sites are hosted by data centers and links between centers are dedicated. As it turns out, our protocol is quite adaptable to different link behaviors.

Servers communicate with each other through the wide-area network to implement a replicated state machine with 1-copy serializability consistency. Servers can fail by crashing, and perhaps later recovering. The system is asynchronous, in that servers and communication do not need to be timely. Clients access the service by sending requests to their local server via local-area communication. We assume it is acceptable for clients not to make progress while their server is crashed. We discuss relaxing this assumption in Section 9.

Consensus is a fundamental coordination problem that requires a group of processes to agree on a common output, based on their (possibly conflicting) inputs. To implement the replicated state machine, the servers run an unbounded sequence of concurrent instances of consensus [33]. Upon receiving a request from a local client, a server assigns the request (*proposes* a value) using one of the unused consensus instances. Multiple servers may propose different values to the same instance of consensus. All *correct* servers (servers that do not crash) eventually agree on a unique request for each used instance, and this request must have been proposed. When servers agree upon a request for a consensus instance, we say that this request has been *chosen*. Note that choosing a request does not imply that the servers know the outcome of the consensus instance. A server *commits* a request once it *learns* the outcome of the consensus instance. Upon commit, the server requests the application service process to execute the request. If the server is the one that originated the request, then it sends the result back to the client. In addition, a server commits an instance only when it has learned and committed all previous consensus instances. [2]

It is straightforward to see that all correct servers eventually learn and execute the same sequence of requests. If the servers do not skip instances when proposing requests, this sequence also contains no gaps. Thus, if all servers start from the same initial state and the service is deterministic, then the service state will always be consistent across servers and servers will always generate consistent responses.

An efficient implementation of replicated state machines should have high throughput under high client load and low latency under low client load.

For throughput, there are two possible bottlenecks in this service, depending upon the average request size:

Wide-area channels When the average request size is large enough, channels saturate before the servers reach their CPU limit. Therefore, the throughput is determined by how efficiently the protocol is able to propagate requests from its originator to the remaining sites. In this case, we say the system is *network-bound*.

Server processing power When the average request size is small enough, the servers reach their CPU limit first. Therefore, the throughput is determined by the processing efficiency at the bottleneck server. In this case, we say the system is *CPU-bound*.

As a rule of thumb, lower message complexity leads to higher throughput because more network bandwidth is available to send actual state machine commands, and less messages per request are processed.

Servers exchange messages to choose and learn the consensus outcome. Each exchange constitutes a *communication step*. To achieve low latency, it is important to have short chains of wide-area communication

steps for the servers to learn the outcome. However, the number of communication steps may not be the only factor impacts latency: high variance on the delivery of message in wide-area networks is also a major contributor [18].

3 Why not Paxos?

Paxos [21, 22] is an efficient asynchronous consensus protocol for replicated state machines. Paxos is a leader-based protocol: one of the servers acts differently than the others, and coordinates the consensus instance. There can be more than one leader at the same time, but during such periods the protocol may not make progress.

Figure 1 illustrates the message flow in a run of a sequence of Paxos instances. Although we show the instances executing sequentially, in practice they can overlap. Each instance of Paxos consists of one or more rounds, and each round can have three phases. Phase 1 (explained in the next paragraph) is only run when there is a leader change. Phase 1 can be simultaneously run for an unbounded number of future instances, which amortizes its cost across all instances that successfully choose a command. Assuming no failures, each server forwards its requests to the leader, which proposes commands (Instance 1 in Figure 1). When the leader receives a proposal, it starts Phase 2 by sending PROPOSE messages (Instance 0 and 1 in Figure 1) that ask the servers (*acceptors* in Paxos terminology) to accept the value. If there are no other leaders concurrently proposing requests, then the servers acknowledge the request with ACCEPT messages. Once the leader receives ACCEPT messages from a majority of the servers, it learns that the value has been chosen and broadcasts a Phase 3 LEARN message to inform the other servers of the consensus outcome. Phase 3 can be omitted by broadcasting ACCEPT messages, which reduces the learning latency for non-leader servers. This option, however, increases the number of messages significantly and so lowers throughput.

When a leader crashes (Instance 2 in Figure 1), the crash is eventually suspected, and another server eventually arises as the new leader. The new leader then starts a higher numbered round and polls the other servers to determine possible commands to propose by running Phase 1 of the protocol. It does this by sending out PREPARE messages and collecting ACK messages from a majority of the servers. Upon finishing Phase 1, the new leader starts Phase 2 to finish any Paxos instances that have been started but not finished by the old leader before crashing.

There are other variants of this protocol, such as Fast Paxos [25] and CoReFP [13], designed to achieve lower latency. Paxos, however, is in general a better candidate for multi-site systems than Fast Paxos and CoReFP because of its simplicity and lower wide-area message complexity, consequently achieving higher throughput.

In the remainder of the paper, we hence compare performance relative only to Paxos. Paxos, however, is still not ideal for wide-area systems:

Unbalanced communication pattern With Paxos, the leader generates and consumes more traffic than the other servers. Figure 1, shows that there is network traffic from replicas to the leader, but no traffic between non-leader replicas. Thus, in a system where sites are pairwise connected, Paxos uses only the channels incident upon the leader, which reduces its ability to sustain high throughput. In addition, during periods of synchrony, only the FWD and PROPOSE messages in Paxos carry significant payload. When the system is network-bound, the volume of these two messages determines the system throughput. In Paxos, FWD is sent from the originator to the leader and PROPOSE is broadcast by the leader. Under high load, the outgoing bandwidth of the leader is a bottleneck, whereas the channels between the non-leaders idle. In contrast, Mencius uses a rotating leader scheme. This not only eliminates the need to send FWD messages, but also gives a more balanced communication pattern, which better utilizes available bandwidth.

Computational bottleneck at the leader The leader in Paxos is a potential bottleneck because it processes more messages than other replicas. When CPU-bound, a system running Paxos reaches its peak capacity when the leader is at full CPU utilization. As the leader requires more processing power than the other servers, the CPU utilization on non-leader servers do not reach their maximum capacity, thus underutilizing the overall processing capacity of the system. The number of messages a leader needs to process for every request grows linearly with the number of servers n, but it remains constant for other replicas. This seriously impacts the scalability of Paxos for larger n. By rotating the leader in Mencius, no single server is a potential bottleneck when the workload is evenly distributed across the sites of the system.

Higher learning latency for non-leader servers While the leader always learns and commits any value it proposes in two communication steps, any other server needs two more communication steps to learn and commit the value it proposes due to the FWD and LEARN messages. With a rotating leader scheme, any server can propose values as a leader. By skipping turns opportunistically when a server has no value to propose, one can achieve the optimal commit delay of two communication steps for any server when there are no concurrent proposals [23]. Concurrent proposals can result in additional delay to commit, but such delays do not always occur. When they do, one can take advantage of commutable operations by having servers execute commands possibly in different, but equivalent orders [24].

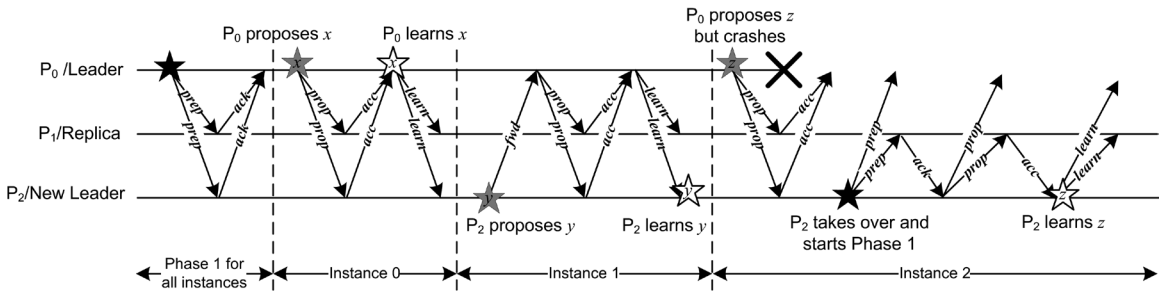


Figure 1: A space time diagram showing the message flow of a sequence of Paxos instances.

4 Deriving Mencius

In this section, we first explain our assumptions and design decisions. We then introduce the concept of simple consensus and use Coordinated Paxos to implement a simple replicated state machine protocol. Finally, we optimize the initial protocol to derive a more efficient one. This last protocol is the one that we call Mencius.

This development of Mencius has two benefits. First, by deriving Mencius from Paxos, Coordinated Paxos, and a set of optimizations and accelerators, it is easier to see that Mencius is correct. Second, one can continue to refine Mencius or even derive a new version of Mencius to adapt it to a particular application.

4.1 Assumptions

We make the following assumptions about the system. We omit a formal description of the assumptions, and we refer readers to [9, 10, 21, 22] for details.

Crash process failure Like Paxos, Mencius assumes that servers fail by crashing and can later recover. Servers have access to stable storage, which they use to recover their states prior to failures.

Unreliable failure detector Consensus is not solvable in an asynchronous environment when even a single process can fail [14]. Like many other asynchronous consensus protocols, Mencius utilizes a failure detector oracle to circumvent the impossibility result. Like Paxos, it relies on the failure detector only for liveness – Mencius is safe even when the failure detector makes an unbounded number of mistakes. Mencius requires that, eventually, all faulty servers and only faulty servers are suspected by the failure detector. In practice, such a failure detector can be implemented by increasing timeouts exponentially. A discussion on our requirements on failure detector can be found in [29].

Asynchronous FIFO communication channel Since we use TCP as the underlying transport protocol, we assume FIFO channels and that messages between two correct servers are eventually delivered. This is a strictly stronger assumption compared to the one of Paxos. Had we instead decided to use UDP, we would have to implement our own message retransmission and flow control at the application layer. Assuming FIFO enables optimizations discussed in Section 4.4. These optimizations, however, are applicable only if both parties of a channel are available and a TCP connection is established. When servers fail and recover after long periods, implementing FIFO channels is impractical as it may require buffering a large number of messages. Mencius uses a separate recovery mechanism that does not depend on FIFO channels (see Section 4.5).

4.2 Simple consensus and Coordinated Paxos

As explained in Section 3, Paxos only allows the leader to propose values. We instead have servers take turns in proposing values. By doing so, servers do not contend when proposing values if there are no failures and no false suspicions. We take advantage of this fact with *simple consensus*.

Simple consensus is consensus in which the values a server can propose are restricted. Let *no-op* be a state machine command that leaves the state unchanged and that generates no response. In simple consensus, only one special server, which we call the *coordinator*, can propose any command (including *no-op*); the others can only propose *no-op*.[3] With Mencius, a replicated state machine runs concurrent instances of simple consensus.

For each instance, one server is designated as the coordinator. Also, the assignment scheme of instances to coordinators is known by all servers. To guarantee that every server has a turn to propose a value, we require that: (1) every server is the coordinator of an unbounded number of instances, and (2) for every server p there is a bounded number of instances assigned to other servers between consecutive instances that p coordinates. A simple scheme assigns instance $cn + p$ to server p, where $c \in \mathbb{N}_0$ and $p \in \{0, \ldots, n-1\}$. Without loss of generality, we assume this scheme for the rest of this paper.

A benefit of using simple consensus is that servers can learn a skipped *no-op* without having to have a majority of servers to agree on it first. As a result, SKIP messages have the minimal learning latency of just one one-way message delay. This ability combined with two optimizations discussed in Section 4.4 makes it possible for the servers to propose *no-op* at very little cost of both com-

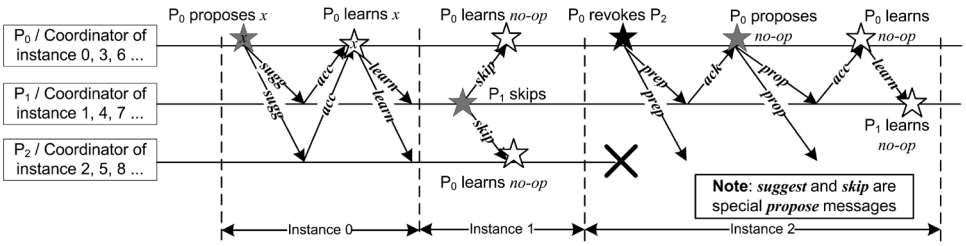


Figure 2: The message flow of suggest, skip and revoke in Coordinated Paxos.

munication and computation overhead. This gives Mencius the ability to adapt quickly and cheaply to changing client load and network bandwidth. Another benefit of simple consensus is discussed in Section 5: by restricting the values a non-coordinator can propose, one can implement a flexible commit mechanism that further reduces Mencius's latency.

Since simple consensus only restricts the initial value a server can propose, any implementation of consensus, including Paxos, can be used to solve simple consensus.

We use, however, an efficient variant of Paxos to implement simple consensus. We call it *Coordinated Paxos* (see [29] for the proof of correctness and the pseudo code). In each instance of Coordinated Paxos, all servers agree that the coordinator is the default leader, and start from the state in which the coordinator had run Phase 1 for some initial round r. Such a state consists of a promise not to accept any value for any round smaller than r. A server can subsequently initiate the following actions, as shown in Figure 2:

Suggest The coordinator *suggests* a request v by sending PROPOSE messages with payload v in round r (Instance 0 in Figure 2). We call these PROPOSE messages SUGGEST messages.

Skip The coordinator *skips* its turn by sending PROPOSE messages that proposes *no-op* in round r (Instance 1 in Figure 2). We call these PROPOSE messages SKIP messages. Note that because all other servers can only propose *no-op*, when the coordinator proposes *no-op*, any server learns that *no-op* has been chosen as soon as it receives a SKIP message from the coordinator.

Revoke When suspecting that the coordinator has failed, some server will eventually arise as the new leader and revoke the right of the coordinator to propose a value. The new leader does so by trying to finish the simple consensus instance on behalf of the coordinator (Instance 2 in Figure 2). Just like a new Paxos leader would do, it starts Phase 1 for some round $r' > r$. If Phase 1 indicates no value may have been chosen, then the new leader proposes *no-op* in Phase 2. Otherwise, it proposes the possible consensus outcome indicated by Phase 1.

The actions *suggest*, *skip* and *revoke* specialize mechanisms that already exist in Paxos. Making them explicit, however, enables more efficient implementations in wide-area networks. The main differences between Coordinated Paxos and Paxos are the following: (1) Coordinated Paxos starts from a different (but safe) state; and (2) a server learns *no-op* upon receiving a SKIP message from the coordinator, and can act accordingly.

4.3 A simple state machine

We now construct an intermediate protocol $\mathcal{P}$ that implements replicated state machines. At high level, $\mathcal{P}$ runs an unbounded sequence of simple consensus instances and each instance is solved with Coordinated Paxos. We describe $\mathcal{P}$ using four rules that determine the behavior of a server and argue that $\mathcal{P}$ is correct using these rules. The pseudo code of $\mathcal{P}$ is in [29]. In Section 4.4, we derive Mencius from $\mathcal{P}$.

While consistency (safety) is guaranteed by Coordinated Paxos, $\mathcal{P}$ still needs to handle duplicate requests that arise from clients submitting requests multiple times due to timeouts. This can be done by using any well-known technique, such as assuming idempotent requests or by recording committed requests and checking for duplicates before committing. We assume such a technique is used.

For liveness, we use Rule 1-4 to ensure that any client request sent to a correct server eventually commits. To minimize the delay in learning, a server suggests a value immediately upon receiving it from a client.

Rule 1 Each server p maintains its next simple consensus sequence number I_p. We call I_p the *index* of server p. Upon receiving a request from a client, a server p suggests the request to the simple consensus instance I_p and updates I_p to the next instance it will coordinate.

Rule 1 by itself performs well only when all servers suggest values at about the same rate. Otherwise, the index of a server generating requests more rapidly will increase faster than the index of a slower server. Servers cannot commit requests before all previous requests are committed, and so Rule 1 commits requests at the rate of the slowest server. In the extreme case that a server suggests no request for a long period of time, the state machine stalls, preventing a potentially unbounded number of requests from committing. Rule 2 uses a technique similar to logical clocks [20] to overcome this problem.

Rule 2 If server p receives a SUGGEST message for instance i and $i > I_p$, before accepting the value and sending back an ACCEPT message, p updates I_p such that its new index $I'_p = min\{k : p \text{ coordinates instance } k \wedge k > i\}$. p also executes skip actions for each of the instances in range $[I_p, I'_p)$ that p coordinates.

With Rule 2, slow servers skip their turns. Consequently, the requests that fast servers suggest do not have to wait for slow servers to have requests to suggest before committing. However, a crashed server does not broadcast SKIP messages, and such a server can prevent others from committing. Rule 3 overcomes this problem.

Rule 3 Let q be a server that another server p suspects has failed, and let C_q be the smallest instance that is coordinated by q and not learned by p. p revokes q for all instances in the range $[C_q, I_p]$ that q coordinates. [4]

If any correct server p suggests a value v to instance i, a server updates its index to a value larger than i upon receiving this SUGGEST message. Thus, according to Rule 2, every correct server r eventually proposes a value (either by skipping or by suggesting) to every instance smaller than i that r coordinates, and all non-faulty servers eventually learn the outcome of those instances. For instances that faulty servers coordinate, according to Rule 3, non-faulty servers eventually revoke them, and non-faulty servers eventually learn the outcome. Thus, all instances prior to i are eventually learned, and request v eventually commits, assuming that p is not falsely suspected by other servers.

False suspicions, however, are possible with unreliable failure detectors. We add Rule 4 to allow a server to suggest a request multiple times upon false suspicions.

Rule 4 If server p suggests a value $v \neq$ *no-op* to instance i, and p learns that *no-op* is chosen, then p suggests v again.

As long as p is not permanently falsely suspected, p will continue to re-suggest v, and v will be eventually chosen. In practice, a period of no false suspicions only needs to hold long enough for p to re-suggest v and have it chosen for the protocol to make progress.

Due to space constraints, we omit the proof of correctness for $\mathcal{P}$ and refer interested readers to [29].

4.4 Optimizations

Protocol $\mathcal{P}$ is correct but not necessarily efficient. It always achieves the minimal two communication steps for a proposing server to learn the consensus value, but its message complexity varies depending on the rates at which the servers suggest values. The worst case is when only one server suggests values, in which case the message complexity is $(n-1)(n+2)$ due to the broadcast SKIP messages that Rule 2 generates.

Consider the case where server p receives a SUGGEST message for instance i from server q. As a result, p skips all of its unused instances smaller than i (Rule 2). Let the first instance that p skips be i_1 and the last instance p skips be i_2. Since p needs to acknowledge the SUGGEST message of q with an ACCEPT message, p can piggyback the SKIP messages on the ACCEPT message. Since channels are FIFO, by the time q receives this ACCEPT message, q has received all the SUGGEST messages p sent to q before sending the ACCEPT message to q. This means that p does not need to include i_1 in the ACCEPT message: i_1 is the first instance coordinated by p that q does not know about. Similarly, i_2 does not need to be included in the ACCEPT message because i_2 is the largest instance smaller than i and coordinated by p. Since both i and p are already included in the ACCEPT message, there is no need for any additional information: all we need to do is augmenting the semantics of the ACCEPT message. In addition to acknowledging the value suggested by q, this message now implies a promise from p that it will not suggest any client requests to any instances smaller than i in the future. This gives us the first optimization:

Optimization 1 p does not send a separate SKIP message to q. Instead, p uses the ACCEPT message that replies the SUGGEST to promise not to suggest any client requests to instances smaller than i in the future.

Protocol $\mathcal{P}$ with Optimization 1 implements replicated state machines correctly.

We can also apply the same technique to the SKIP messages from p to other servers. Instead of using ACCEPT messages, we piggyback the SKIP messages on future SUGGEST messages from p to another server r:

Optimization 2 p does not send a SKIP message to r immediately. Instead, p waits for a future SUGGEST message from p to r to indicate that p has promised not to suggest any client requests to instances smaller than i.

Note that Optimization 2 can potentially defer the propagation of SKIP messages from p to r for an unbounded period of time. For example, consider three servers p_0, p_1, p_2. Only p_0 suggests values for instance 0, 3, 6, and so on. p_0 always learns the result for all instances by means of the ACCEPT messages from p_1 and p_2. Server p_1, however, learns all values that p_0 proposes, and it knows which instances it is skipping, but it does not learn that p_2 skips, such as for instance 2 in this example. This leaves gaps in the view of p_1 of the consensus sequence and prevents p_1 from committing values learned in instance 3, 6, and so on. Similarly, p_2 does not learn that p_1 is skipping and prevents p_2 from committing values learned in 3, 6, and so on.

This problem only occurs between two idle servers p_1 and p_2: any value suggested by either server will propagate the SKIP messages in both directions and hence fill in the gaps. Fortunately, while idle, neither p_1 nor p_2 is responsible for generating replies to the clients. This

means that, from the client perspective, its individual requests are still being processed in a timely manner, even if p_1 and p_2 are stalled. We use a simple accelerator rule to limit the number of outstanding SKIP messages before p_1 and p_2 start to catch up:

Accelerator 1 A server p propagates SKIP messages to r if the total number of outstanding SKIP messages to r is larger than some constant α, or the messages have been deferred for more than some time τ.

Note that Optimization 2 and Accelerator 1 can only delay the propagation of SKIP messages for a bounded amount of time. Since $\mathcal{P}$ only relies on the eventual delivery of messages for liveness, adding Optimization 2 and Accelerator 1 to protocol $\mathcal{P}$ still implements replicated state machines correctly.

Given that the number of extra SKIP messages generated by Accelerator 1 are negligible over the long run, the amortized wide-area message complexity for Mencius is $3n-3$ ($(n-1)$ SUGGEST, ACCEPT and LEARN messages each), the same as Paxos when FWD is not considered.

We can also reduce the extra cost generated by the revocation mechanism. If server q crashes, revocations need to be issued for every simple consensus instance that q coordinates. By doing this, we increase both the latency and message complexity due to the use of the full three phases of Paxos. A simple idea is to revoke all q's future turns, which irreversibly chooses *no-op* for all q's further turns. However, q may need to suggest values in the future, either because q was falsely suspected or because it recovers. A better idea is the following:

Optimization 3 Let q be a server that another server p suspects has failed, and let C_q be the smallest instance that is coordinated by q and not learned by p. For some constant β, p revokes q for all instances in the range $[C_q, I_p + 2\beta]$ that q coordinates if $C_q < I_p + \beta$.

Optimization 3 allows p to revoke q at least β instances in advance before p suggests a value to some instance i greater than C_q. By tuning β, we ensure that by the time p learns the outcome of instance i, all instances prior to i and coordinated by q are revoked and learned. Thus, p can commit instance i without further delay. Since Optimization 3 also requires revocations being issued in large blocks, the amortized message cost is small.

Note that Optimization 3 can only exclude the actions of a falsely suspected server for a bounded number of instances. Since we assume the failure detector will eventually be accurate, such false suspicions will eventually cease. So, using Optimization 3 does not affect the liveness of the protocol.

Optimization 3 addresses the common case where there are no false suspicions. When a false suspicion does occur, it may result in poor performance while servers are falsely suspected. We consider the poor performance in this case acceptable because we assume false suspicions occur rarely in practice and the cost of recovery from a false suspicion is small (see Section 6).

Mencius is $\mathcal{P}$ combined with Optimizations 1-3 and Accelerator 1. From the above arguments, Mencius implements replicated state machines correctly. Due to lack of space, we omit the proof of correctness and the pseudo code, both of which can be found in [29].

Mencius, being derived from Paxos, has the same quorum size of $f+1$. This means that up to f servers can fail among a set of $2f+1$ servers. Paxos incurs temporarily reduced performance when the leader fails. Since all servers in Mencius act as a leader for an unbounded number of instances, Mencius has this reduced performance when *any* server fails. Thus, Mencius has higher performance than Paxos in the failure-free case at the cost of potentially higher latency upon failures. Note that higher latency upon failures also depends on other factors such as the stability of the communication network.

4.5 Recovery

In this section, we outline how Mencius recovers from failures. Due to lack of space, we omit the details.

Temporary broken TCP connection We add an application layer sequence number to Mencius's messages. FIFO channels are maintained by retransmitting missing messages upon reestablishing the TCP connection.

Short term failure Like Paxos, Mencius logs its state to stable storage and recovers from short term failures by replaying the logs and learning recent chosen requests from other servers.

Long term failure It is impractical for a server to recover from a long period of down time by simply learning missing sequences from other servers, since this requires correct servers to maintain an unbounded long log. The best way to handle this, such as with checkpoints or state transfer [8, 27], is usually application specific.

5 Commit delay and out-of-order commit

In Paxos, the leader serializes the requests from all the servers. For purposes of comparison, assume that Paxos is implemented, like Mencius, using FIFO channels. If the leader does not crash, then each server learns the requests in order, and can commit a request as soon as it learns the request. The leader can commit a request as soon as it collects ACCEPT messages from a quorum of $f+1$ servers, and any other server will have an additional round trip delay due to the FWD and LEARN messages.

While a Mencius server can commit the request in just one round trip delay when there is no contention, commits may have to be delayed up to two communication steps when there are concurrent suggestions.

For example, in the scenario illustrated in Figure 3, server p_0 suggests x to instance 0 concurrently with p_1 suggesting y to instance 1. p_1 receives the SUGGEST

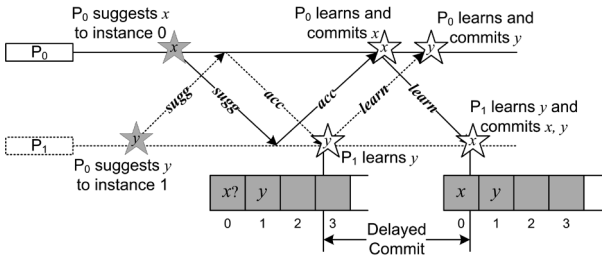


Figure 3: Delayed commit.

message for x from p_0 before it receives the ACCEPT message for y. Upon receiving the ACCEPT for y from p_0, p_1 learns that y has been chosen for instance 1, but cannot commit y yet as it has only accepted but not learned x for instance 0. In this case, p_1 cannot commit y until it receives the LEARN message for x from p_0, at which point it can commit both x and y at once. We say that y experiences a *delayed commit* at p_1.

The delay can be up to two communication steps, since p_1 must learn y in between accepting x and learning x for a delayed commit to occur. If p_1 learns y after it learns x, then there is clearly no extra delay. If p_1 learns y before it accepts x, then p_0 must have accepted y before suggesting x because of the FIFO property of the channel. In this case, according to Rule 2, p_0 must have skipped instance 0, which contradicts the assumption that p_0 suggested x to instance 0. Thus, the extra delay to commit y can be as long as one round trip communication between p_0 and p_1 (p_1 sends ACCEPT to p_0 and p_0 sends LEARN back), *i.e.*, up to two communication steps. We can reduce the upper bound of delayed commit to one communication step by broadcasting ACCEPT messages and eliminating LEARN messages. This reduction gives Mencius an optimal commit delay of three communication steps when there are concurrent proposals [23] at the cost of higher message complexity and thus lower throughput.

Because delayed commit arises with concurrent suggestions, it becomes more of a problem as the number of suggestions grows. In addition, delayed commit impacts the commit latency but not overall throughput: over a long period of time, the total number of requests committed is independent of delayed commits.

Out-of-order commit We can mitigate the effects of delayed commit with a simple and more flexible commit mechanism that allows x and y to be executed in any order when they are *commutable*, *i.e.*, executing x followed by y produces the same system state as executing y followed by x. By the definition of simple consensus, when p_1 receives the SUGGEST message for x, it knows that only x or *no-op* can be chosen for instance 0. Since *no-op* commutes with any request, upon learning y, p_1 can commit y before learning x and send the result back to the client without any delay when x and y are commutable. We call this mechanism *out-of-order commit* and evaluate its effectiveness in Section 7.5. We implement out-of-order commit in Mencius by tracking the dependencies between the requests and by committing a request as soon as all requests it depends on have been committed. This technique can not be applied to Paxos as easily, because Paxos is based on consensus, which does not have any restriction on the value a server can propose – the key for Mencius to guarantee safety while allowing out-of-order commit.

6 Choosing parameters

Accelerator 1 and Optimization 3 use three parameters: α, β and τ. We discuss here strategies for choosing these parameters.

Accelerator 1 limits the number of outstanding SKIP messages between two idle server p_1 and p_2 before they start to catch up. It bounds both the amount of time (τ) and number of outstanding messages (α).

When choosing τ, it should be large enough so that the cost of SKIP messages can be amortized. But, a larger τ adds more delay to the propagation of SKIP messages, and so results in extra commit delay for requests learned at p_1 and p_2. Fortunately, when idle, neither p_1 nor p_2 generates any replies to the clients, and so such extra delay has little impact from a client's point of view. For example, in a system with 50 ms one-way link delay, we can set τ to the one-way delay. This is a good value because: (1) With $\tau = 50$ ms, Accelerator 1 generates at most 20 SKIP messages per second, if α is large enough. The network resource and CPU power needed to transmit and process these messages are negligible; and (2) The extra delay added to the propagation of the SKIP messages is at most 50 ms, which could occur anyway due to network delivery variance or packet loss.

α limits the number of outstanding SKIP messages before p_1 and p_2 start to catch up: if τ is large enough, α SKIP messages are combined into just one SKIP message, reducing the overhead of SKIP messages by a factor of α. For example, we set α to 20 in our implementation, which reduces the cost of SKIP message by 95%.

β defines an interval of instances: if a server q is crashed and I_p is the index of a non-faulty server p, then in steady state all instances coordinated by q and in the range $[I_p, I_p + k]$ for some $k : \beta \leq k \leq 2\beta$ are revoked. Choosing a large β guarantees that while crashed, q's inactivity will not slow down other servers. It, however, makes the indexes of q and other servers more out of synchronization when q recovers from a false suspicion or a failure. Nonetheless, the overhead of having a large β is negligible. Upon recovery, q will learn the instances it coordinates that have been revoked. It then

updates its index to the next available slot and suggests the next client request using that instance. Upon receiving the SUGGEST message, other replicas skip their turns and catch up with q's index (Rule 2). The communication overhead of skipping is small, as discussed in Optimization 1 and 2. The computation overhead of skipping multiple consecutive instances at once is also small, since an efficient implementation can easily combine their states and represent them at the cost of just one instance. While setting β too large could introduce problems with consensus instance sequence number wrapping, any practical implementation should have plenty of room to choose an appropriate β.

Here is one way to calculate a lower bound for β. Revocation takes up to two and a half round trip delays. Let i be an instance of server q that is revoked. To avoid delayed commit of some instance $i' > i$ at a server p, one needs to start revoking i two and a half round trips in advance of instance i' being learned by p. In our implementation with a round trip delay of 100 ms and with $n = 3$, the maximum throughput is about $10,000$ operations per second. Two and a half round trip delays are 250 ms, which, at maximum throughput, is $2,500$ operations. All of these operations could be proposed by a single server, and so the instance number may advance by as many as $3 \times 2,500 = 7,500$ in any 250 ms interval. Thus, if $\beta \geq 7,500$, then in steady state no instances will suffer delayed commit arising from q being crashed. Taking network deliver variance into account, we set $\beta = 100,000$, which is a conservative value that is more than ten times the lower bound, but still reasonably small even for the 32-bit sequence number space in our implementation.

7 Evaluation

We ran controlled experiments in the DETER testbed [5] to evaluate the performance of Mencius and Paxos. We used TCP as the transport protocol and implemented both protocols in C++. Here are some implementation details:

API Both Paxos and Mencius implement two simple API calls: PROPOSE(v) and ONCOMMIT(v). An application calls PROPOSE to issue a request, and the state machine upcalls the application via ONCOMMIT when the request is ready to commit. When out-of-order commit is enabled, Mencius uses a third upcall ISCOMMUTE(u, v) to ask the application if two requests are commutable.

Nagle's algorithm Nagle's algorithm [30] is a technique in TCP for improving the efficiency of wide-area communication by batching small messages into larger ones. It does so by delaying sending small messages and waiting for data from the application. In our implementation, we can instruct servers to dynamically turn on or turn off Nagle's algorithm.

Parameters We set the parameters that control Accelerator 1 and Optimization 3 to $\alpha = 20$ messages, $\tau = 50$ ms, and $\beta = 100,000$ instances.

7.1 Experimental settings

To compare the performance of Mencius and Paxos, we use a simple, low-overhead application that enables commutable operations. We chose a simple read/write register service of κ registers. The service implements a read and a write command. Each command consists of the following fields: (1) operation type – read or write (1 bit); (2) register name (2 bytes); (3) the request sequence number (4 bytes); and (4) ρ bytes of dummy payload. All the commands are ordered by the replicated state machine in our implementation. When a server commits a request, it executes the action, sends a zero-byte reply to the client and logs the first three fields along with the client's ID. We use the logs to verify that all servers learn the same client request sequence; or, when reordering is allowed, that the servers learned compatible orders. Upon receiving the reply from the server, the client computes and logs the latency of the request. We use the client-side log to analyze experiment results.

We evaluated the protocols using a three-server clique topology for all but the experiments in Section 7.4. This architecture simulated three data centers (A, B and C) connected by dedicated links. Each site had one server node running the replicated register service, and one client node that generated all the client requests from that site. Each node was a 3.0 GHz Dual-Xeon PC with 2.0 GB memory running Fedora 6. Each client generated requests at either a fixed rate or with inter-request delays chosen randomly from a uniform distribution. The additional payload size ρ was set to be 0 or $4,000$ bytes. 50% of the requests were reads and 50% were writes. The register name was uniformly chosen from the total number of registers the service implemented. A virtual link was set up between each pair of sites using the DummyNet [31] utility. Each link had a one-way delay of 50 ms. We also experimented with other delay settings such as 25 ms and 100 ms, but do not report these results here because we did not observe significant differences in the findings. The link bandwidth values varied from 5 Mbps to 20 Mbps. When the bandwidths were chosen within this range, the system was network-bound when $\rho = 4,000$ and CPU-bound when $\rho = 0$. Except where noted, Nagle's algorithm was enabled.

In this section, we use "Paxos" to denote the register service implemented with Paxos, "Mencius" to denote the register service using Mencius and with out-of-order commit disabled, and "Mencius-κ" to denote the service using Mencius with κ total registers and out-of-order commit enabled (*e.g.*, Mencius-128 corresponds to the service with 128 registers). Given the read/write ratio, requests in Mencius-κ can be moved up, on average,

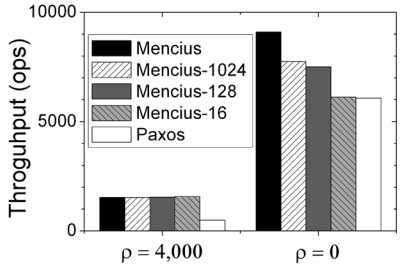


Figure 4: Throughput for 20 Mbps bandwidth

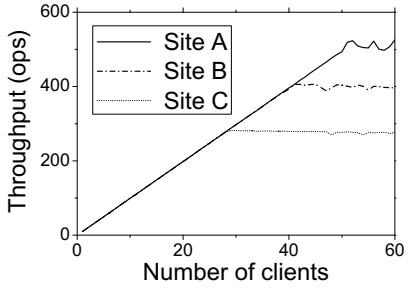


Figure 5: $\rho = 4,000$ with asymmetric bandwidth

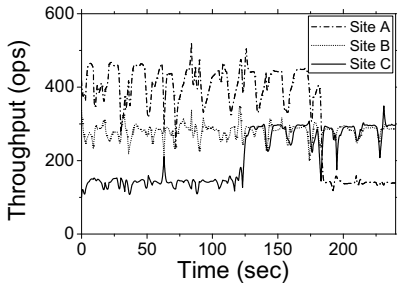


Figure 6: Mencius dynamically adapts to changing network bandwidth ($\rho = 4,000$)

0.75κ slots before reaching an incommutable request. We used κ equal to 16, 128, or $1,024$ registers to represent a service with low, moderate and a high likelihood of adjacent requests being commutable, respectively.

We first describe, in Section 7.2, the throughput of the service both when it is CPU-bound and when it is network-bound, and we show the impact of asymmetric channels and variable bandwidth. In both cases, Mencius has higher throughput. We further evaluate both protocols under failures in Section 7.3. In Section 7.4 we show that Mencius is more scalable than Paxos. In Section 7.5 we measure latency and observe the impact of delayed commit. In general, as load increases, the commit latency of Mencius degrades from being lower than Paxos to being the same as the one of Paxos. Reordering requests decreases the commit latency of Mencius. Finally, we show that the impact of variance in network latency is complex.

7.2 Throughput

To measure throughput, we use a large number of clients generating requests at a high rate. Figure 4 shows the throughput of the protocols, for a fully-connected topology with 20 Mbps available for each link, and a total of 120 Mbps available bandwidth for the whole system.

When $\rho = 4,000$, the system was network-bound: all four Mencius variants had a fixed throughput of about $1,550$ operations per sec (ops). This corresponds to 99.2 Mbps, or 82.7% utilization of the total bandwidth, not counting the TCP/IP and MAC header overhead. Paxos had a throughput of about 540 ops, or one third of Mencius's throughput: Paxos is limited by the leader's outgoing bandwidth.

When $\rho = 0$, the system is CPU-bound. Paxos presents a throughput of $6,000$ ops, with 100% CPU utilization at the leader and 50% at the other servers. Mencius's throughput under the same condition was $9,000$ ops, and all three servers reached 100% CPU utilization. Note that the throughput improvement for Mencius was in proportion to the extra CPU processing power available. Mencius with out-of-order commit enabled had lower throughput compared to Mencius with this feature disabled because Mencius had to do the extra work of dependency tracking. The throughput drops as the total number of registers decreases because with fewer registers there is more contention and dependencies to handle.

Figure 5 demonstrates Mencius's ability to use available bandwidth even when channels are asymmetric with respect to bandwidth. Here, we set the bandwidth of the links $A \rightarrow B$ and $A \rightarrow C$ to 20 Mbps, links $B \rightarrow C$ and $B \rightarrow A$ to 15 Mbps and links $C \rightarrow B$ and $C \rightarrow A$ to 10 Mbps. We varied the number of clients, ensuring that each site had the same number of clients. Each client generated requests at a constant rate of 100 ops. The additional payload size ρ was $4,000$ bytes. As we increased the number of clients, site C eventually saturated its outgoing links first; and from that point on committed requests at a maximum throughput of 285 ops. In the meanwhile, the throughput at both A and B increased until site B saturated its outgoing links at 420 ops. Finally site A saturated its outgoing links at 530 ops. As expected, the maximum throughput at each site is proportional to the outgoing bandwidth (in fact, the minimum bandwidth).

Figure 6, shows Mencius's ability to adapt to changing network bandwidth. We set the bandwidth of links $A \rightarrow B$ and $A \rightarrow C$ to 15 Mbps, links $B \rightarrow A$ and $B \rightarrow C$ to 10 Mbps, and link $C \rightarrow A$ and $C \rightarrow B$ to 5 Mbps. Each site had a large number of clients generating enough requests to saturate the available bandwidth. Site A, B and C initially committed requests with throughput of about 450 ops, 300 ops, and 150 ops respectively, reflecting the bandwidth available to them. At time $t = 60$ seconds,

we dynamically increased the bandwidth of link $C \rightarrow A$ from 5 Mbps to 10 Mbps. With the exception of a spike, C's throughput did not increase because it is limited by the 5 Mbps link from C to B. At $t = 120$ seconds, we dynamically increased the bandwidth of link $C \rightarrow B$ from 5 Mbps to 10 Mbps. This time, site C's throughput doubles accordingly. At $t = 180$ seconds, we dynamically decreased the bandwidth of link $A \rightarrow C$ from 15 Mbps to 5 Mbps. The throughput at site A dropped, as expected, to one third.

In summary, Mencius achieves higher throughput compared to Paxos under both CPU-bound and network-bound workload. Mencius also fully utilizes available bandwidth and adapts to bandwidth changes.

7.3 Throughput under failure

In this section, we show throughput during and after a server failure. We ran both protocols with three servers under network-bound workload ($\rho = 4,000$). After 30 seconds, we crashed one server. We implemented a simple failure detector that suspects a peer when it detects the loss of TCP connection. The suspicion happened quickly, and so we delayed reporting the failure to the suspecting servers for another five seconds. Doing so made it clearer what occurs during the interval when a server's crash has not yet been suspected.

Figure 7(a) shows Mencius's instantaneous throughput observed at server p_0 when we crash server p_1. The throughput is roughly 850 ops in the beginning, and quickly drops to zero when p_1 crashes. During the period the failure remains unreported, both p_0 and p_2 are still able to make progress and learn instances they coordinate, but cannot commit these instances because they have to wait for the consensus outcome of the missing instances coordinated by p_1. When the failure detector reports the failure, p_0 starts revocation against p_1. At the end of the revocation, p_0 and p_2 learn of a large block of *no-op*s for instances coordinated by p_1. This enables p_0 to commit all instances learned during the five second period in which the failure was not reported, which results in a sharp spike of 3,600 ops. Once these instances are committed, Mencius's throughput stabilizes at roughly 580 ops. This is two thirds of the rate before the failure, because there is a reduction in the available bandwidth (there are fewer outgoing links), but it is still higher than that of Paxos under the same condition.

Figure 7(b) shows Paxos's instantaneous throughput observed at server p_1 when we crash the leader p_0. Throughput is roughly 285 ops before the failure, and it quickly drops to zero when p_0 crashes because the leader serializes all requests. Throughput remains zero for five seconds until p_1 becomes the new leader, which then starts recovering previously unfinished instances. Once it finishes recovering such instances, Paxos's throughput goes back to 285 ops, which was roughly the throughput before the failure of p_0. Note that at $t = 45$ seconds, there is a sharp drop in the throughput observed at p_1. This is due to duplicates: upon discovering the crash of p_0, both p_1 and p_2 need to re-propose requests that have been forwarded to p_0 and are still unlearned. Some of the requests, however, have sequence numbers (assigned by p_0) and have been accepted by either p_1 or p_2. Upon taking leadership, p_1 revokes such instances, hence resulting in duplicates. In addition, the throughput at p_1 has higher variance after the failure than before. This is consistent with our observation that the Paxos leader sees higher variance than other servers.

Figure 7(c) shows Paxos's instantaneous throughput of leader p_0 when p_1 crashes. There is a small transient drop in throughput but since the leader and a majority of servers remain operational, throughput quickly recovers.

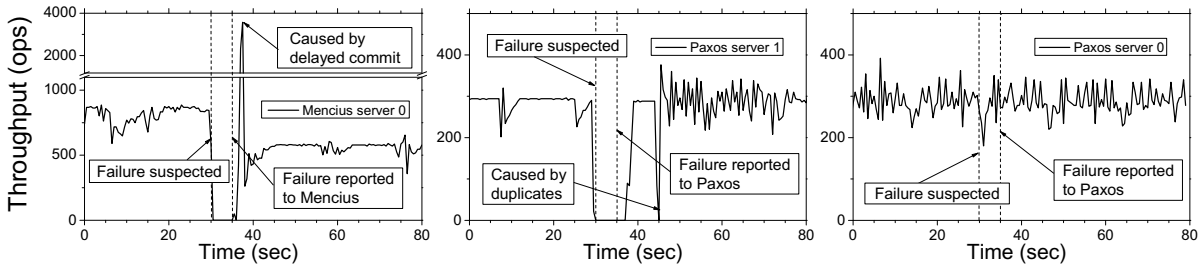
To summarize, Mencius temporarily stalls when any of the servers fails while Paxos temporarily stalls only when the leader fails. Also, the throughput of Mencius drops after a failure because of a reduction on available bandwidth, while the throughput of Paxos does not change since it does not use all available bandwidth.

7.4 Scalability

For both Paxos and Mencius, availability increases by increasing the number of servers. Given that wide-area systems often target an increasing population of users, and sites in a wide-area network can periodically disconnect, scalability is an important property.

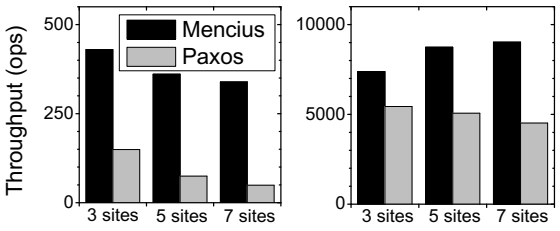
We evaluated the scalability of both protocols by running them with a state machine ensemble of three, five and seven sites. We used a star topology where all sites connected to a central node: these links had a bandwidth of 10 Mbps and 25 ms one-way delay. We chose the star topology to represent the Internet cloud as the central node models the cloud. The 10 Mbps link from a site represents the aggregated bandwidth from that site to all other sites. We chose 10 Mbps because it is large enough to have a CPU-bound system when $\rho = 0$, but small enough so that the system is network-bound when $\rho = 4,000$. When $n = 7$, 10 Mbps for each link gives a maximum demand of 70 Mbps for the central node, which is just under its 100 Mbps capacity. The 25 ms one-way delay to the central node gives an effective 50 ms one-way delay between any two sites. Because we only consider throughput in this section, network latency is irrelevant. To limit the number of machines we use, we chose to run the clients and the server on the same physical machine at each site. Doing this takes away some of the CPU processing power from the server; this is equivalent to running the experiments on slower machines under CPU-bound workload ($\rho = 0$), and has no effect under network-bound workload ($\rho = 4,000$).

When the system is network-bound, increasing the number of sites (n) makes both protocols consume more



(a) One Mencius server crashes (b) Paxos leader crashes (c) One Paxos non-leader crashes

Figure 7: Mencius and Paxos's throughput under failure



(a) $\rho = 4,000$ (b) $\rho = 0$

Figure 8: Throughput vs. number of sites

bandwidth per request: each site sends a request to each of the remaining $n-1$ sites. Since Paxos is limited by the leader's total outgoing bandwidth, its throughput is in proportion to $\frac{1}{n-1}$. Mencius, on the other hand, can use the extra bandwidth provided by the new sites, and so the throughput is in proportion to $\frac{n}{n-1}$. Figure 8(a) shows both protocols' throughput with $\rho = 4,000$. Mencius started with a throughput of 430 ops with three sites, approximately three times higher than Paxos's 150 ops under the same condition. When n increased to five, Mencius's throughput drops to 360 ops ($84\% \approx (\frac{5}{4})/(\frac{3}{2})$), while Paxos's drops to 75 ops ($50\% = (\frac{1}{4})/(\frac{1}{2})$). When n increased to seven, Mencius's throughput dropped to 340 ops ($79\% \approx (\frac{7}{6})/(\frac{3}{2})$) while Paxos's dropped to 50 ops ($33\% = (\frac{1}{6})/(\frac{1}{2})$).

When the system is CPU-bound, increasing n requires the leader to perform more work for each client request. Since the CPU of the leader is a bottleneck for Paxos, its throughput drops as n increases. Mencius, by rotating the leader, takes advantage of the extra processing power. Figure 8(b) shows throughput for both protocols with $\rho = 0$. As n increases, Paxos's throughput decreases gradually. Mencius's throughput increases gradually because more processing power outweighs the increasing processing cost for each request. When $n = 7$, Mencius's throughput is almost double that of Paxos.

7.5 Latency

In this section, we use the three-site clique topology to measure Mencius's commit latency under low to medium load. We ran the experiments with both Nagle on and off. Not surprisingly, both Mencius and Paxos with Nagle on show a higher commit latency due to the extra delay added by Nagle's algorithm. Having Nagle's enabled also adds some variability to the commit latency. For example, with Paxos, instead of a constant commit latency of 100 ms at the leader, the latency varied from 100 to 250 ms with a concentration around 150 ms. Except for this, Nagle's algorithm does not affect the general behavior of commit latency. Therefore, for the sake of clarity, we only present the results with Nagle off for the first two experiments. With Nagle turned off, all experiments with Paxos showed a constant latency of 100 ms at the leader and 200 ms for the other servers. Since we have three servers, Paxos's average latency was 167 ms. In the last set of experiments, we increased the load and so turned Nagle on for more efficient network utilization.

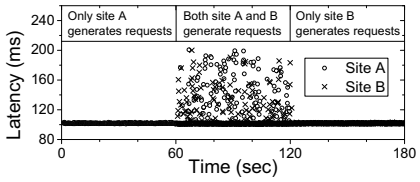


Figure 9: Mencius's commit latency when client load shifts from one site to another

In a wide-area system, the load of different sites can be different for many reasons, such as time zone. To demonstrate the ability of Mencius to adjust to a changing client load, we ran a three-minute experiment with one client on site A and one on B. Site A's client generated requests during the first two minutes and site B's client generated requests during the last two minutes. Both clients generate requests at the same rate ($\delta \in [100\text{ ms}, 200\text{ ms}]$). Figure 9 shows that during the first minute when only site A generated requests, all requests had the minimal 100 ms commit latency. In the next minute when both sites A and B generated requests, the majority of the requests still had the minimal 100 ms delay, but some requests experienced extra delayed commits of up to 100 ms. During the last minute, the laten-

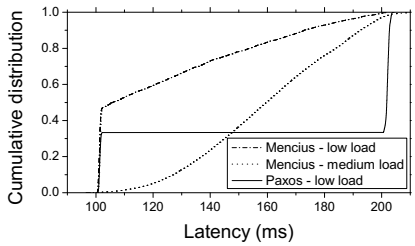


Figure 10: Commit latency distribution under low and medium load

cies return to 100 ms.

To further see the impact of delayed commit, we ran experiments with one client at each site and all three clients concurrently generating requests. Figure 10 plots the CDF of the commit latency under low load (the inter-request delay of $\delta \in [100\text{ ms}, 200\text{ ms}]$) and medium load ($\delta \in [10\text{ ms}, 20\text{ ms}]$). We show only the low load distribution for Paxos because the distribution for medium load is indistinguishable from the one we show. For Paxos, one third of the requests had a commit latency of 100 ms and two thirds had a 200 ms latency. With low load the contention level was low and delayed commit happened less often for Mencius. As a result, about 50% of the Mencius requests have the minimal 100 ms delay. For those requests that did experience delayed commits, the extra latency is roughly uniformly distributed in the range $(0\text{ ms}, 100\text{ ms})$. Under medium load, the concurrency level goes up and almost all requests experience delayed commits. The average latency is about 155 ms, which is still better than Paxos's average of 167 ms under the same condition.

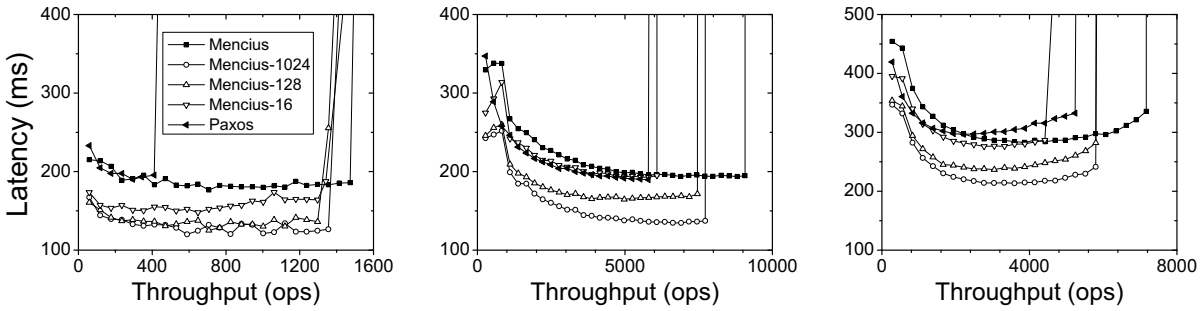
For the experiments of Figure 11, we increased the load by adding more clients, and we enabled Nagle. All curves show lower latency under higher load. This is because of the extra delay introduced by Nagle: the higher the client load, the more often messages are sent, and therefore on average, the less time any individual message is buffered by Nagle. This effect is much weaker in the $\rho = 4,000$ cases than the $\rho = 0$ case because Nagle has more impact on small messages. All experiments also show a rapid jump in latency as the protocols reach their maximum throughput: at this point, the queues of client requests start to grow rapidly.

Figure 11(a) shows the result for the network-bound case of $\rho = 4,000$. Mencius and Paxos had about the same latency before Paxos reached its maximum throughput. At this point, delayed commit has become frequent enough that Mencius has the same latency as Paxos. Lower latency can be obtained by allowing commutable requests to be reordered. Indeed, Mencius-1024, which has the lowest level of contention, had the lowest latency. For example, at 340 ops, Paxos and Mencius showed an average latency of 195 ms, Mencius-16 had an average latency of 150 ms, and Mencius-128 and Mencius-1024 had an average latency of 130 ms, which is an approximate 30% improvement. As client load increased, Mencius's latency remained roughly the same, whereas Mencius-16's latency increased gradually because the higher client load resulted in fewer opportunities to take advantage of commutable requests. Finally, Mencius-128 and Mencius-1024 showed about the same latency as client load increased, with Mencius-1024 being slightly better. This is because at the maximum client load ($1,400$ ops) and correspondent latency (130 ms), the maximum number of concurrently running requests is about 180 requests. This gave Mencius-128 and Mencius-1024 about the same opportunity to reorder requests.

Figure 11(b) shows the result for the CPU-bound case of $\rho = 0$. It shows the same trends as Figure 11(a). The impact of Nagle on latency is more obvious, and before reaching 900 ops, the latency of all four variants of Mencius increases as load goes up. This is because delayed commits happened more often as the load increased. We see the increase in latency because the penalty from delayed commits outweighed the benefits gained by being delayed, on average, for less time by Nagle. In addition, Mencius started with a slightly worse latency than Paxos, and the gap between the two decreased as throughput goes up. Out-of-order commit helps Mencius to reduces its latency: Mencius-16 (a high contention level) had about the same latency as Paxos. Finally, Mencius-128's latency was between Mencius-16 and Mencius-1024. As client load increased, the latency for Mencius-128 tended away from Mencius-1024 towards Mencius-16. This is because the higher load resulted in higher contention: increased contention gave Mencius-128 less and less flexibility to reorder requests.

In the experiment of Figure 11(c), we select delivery latencies at random. It is the same experiment as the one of Figure 11(b), except that we add a Pareto distribution to each link using the NetEm [17] utility. The average extra latency is 20 ms and the variance is 20 ms. The latency time correlation is 50%, meaning that 50% of the latency of the next packet depends on the latency of the current packet. Pareto is a heavy tailed distribution, which models the fact that wide-area links are usually timely but can present high latency occasionally. Given the 20 ms average and 20 ms variance, we observe the extra latency range from 0 to 100 ms. This is at least a twofold increase in latency at the tail. We also experimented with different parameters and distributions, but we do not report them here as we did not observe significant differences in the general trend.

The shapes of the curves in Figure 11(c) are similar to those in Figure 11(b), despite the network variance,



(a) $\rho = 4,000$, no network variance (b) $\rho = 0$, no network variance (c) $\rho = 0$, with network variance

Figure 11: Commit latency vs offered client load

except for the following: (1) All protocols have lower throughput despite the system being CPU-bound – high network variance results in packets being delivered out-of-order, and TCP has to reorder and retransmit packets, since out-of-order delivery of ACK packets triggers TCP fast retransmission. (2) At the beginning of the curves in figure 11(b), all four Mencius variants show lower latency under lower load because delayed commit happened less often. There is no such a trend in Figure 11(c). This happens because with Mencius we wait for both servers to reply before committing a request, whereas with Paxos we only wait for the fastest server to reply. The penalty for waiting for the extra reply is an important factor under low load and results in higher latency for Mencius. For example, at 300 ops, Mencius's latency is 455 ms compared to Paxos's 415 ms delay. However, out-of-order commit helps Mencius to achieve lower latency: Mencius-16 shows 400 ms delay while both Mencius-128 and Mencius-1024 show 350 ms delay. (3) As load increases, Paxos's latency becomes larger than Mencius's. This is due to the higher latency observed at non-leader servers. Although with Paxos the leader only waits for the fastest reply to learn a request, the non-leaders have the extra delay of FWD and LEARN messages. Consider two consecutive requests u and v assigned to instances i and $i+1$, respectively. If the LEARN message for u arrives at a non-leader later than the LEARN message for v because of network variance, the server cannot commit v for instance $i+1$ until it learns u for instance i. If the delay between learning v and learning u is long, then the commit delay of v is also long. Note that in our implementation, TCP causes this delay as TCP orders packets that are delivered out of order. Under higher load, the interval between u and v is shorter, and the penalty instance $i+1$ takes is larger because of the longer relative delay of the LEARN message for instance i.

In summary, Mencius has lower latency than Paxos when network latency has little variance. The out-of-order commit mechanism helps Mencius reduce up to 30% its latency. Non-negligible network variance has negative impact on Mencius's latency under low load, but low load also gives Mencius's out-of-order commit mechanism more opportunity to reduce latency. And, under higher load, Paxos shows higher latency than Mencius because of the impact of network variance on non-leader replicas.

7.6 Other possible optimizations

There are other ways one can increase throughput or reduce latency. One idea is to batch multiple requests into a single message, which increases throughput at the expense of increased latency. This technique can be applied to both protocols, and would have the same benefit. We verified this with a simple experiment: we applied a batching strategy that combined up to five messages that arrive within 50 ms into one. With small messages ($\rho = 0$), Paxos throughput increased by 4.9 and Mencius by 4.8; with large messages the network was the bottleneck and throughput remained unchanged.

An approach to reducing latency consists of eliminating Phase 3 and instead broadcasting ACCEPT messages. This approach cuts for Paxos the learning delay of non-leaders by one communication step, and for Mencius it reduces the upper bound on delayed commit by one communication step. For both protocols, it increases the message complexity from $3n-3$ to n^2-1, thus reducing throughput when the system is CPU-bound. However, doing so has little effect on throughput when the system is network-bound, because the extra messages are small control messages that are negligible compared to the payload of the requests.

Another optimization for Paxos is to have the servers broadcast the body of the requests and reach consensus on a unique identifier for each request. This optimization allows Paxos, like Mencius, to take full advantage of the available link bandwidth when the service is network-bound. It is not effective, however, when the service is CPU-bound, since it might reduce Paxos's throughput by increasing the wide-area message complexity.

8 Related work

Mencius is derived from Paxos [21, 22]. Fast Paxos, one of the variants of Paxos [25], has been designed to improve latency. However, it suffers from collisions (which results in significantly higher latency) when concurrent proposals occur. Another protocol, CoReFP [13], deals with collisions by running Paxos and Fast Paxos concurrently, but has lower throughput due to increased message complexity. Generalized Paxos [24], on the other hand, avoid collisions by allowing Fast Paxos to commit requests in different but equivalent orders. In Mencius, we allow all servers to immediately assign requests to the instances they coordinate to obtain low latency. We avoid contention by rotating the leader (coordinator), which is called a *moving sequencer* in the classification of Défago *et al.* [12]. We also use the rotating leader scheme to achieve high throughput by balancing network utilization. Mencius, like Generalized Paxos, can also commit requests in different but equivalent orders.

Another moving sequencer protocol is Totem [4] which enables any server to broadcast by passing a token. A process in Totem, however, has to wait for the token before broadcasting a message, whereas a Mencius server does not have to wait to propose a request. Lamport's application of multiple leaders [26] is the closest to Mencius. It is primarily used to remove the single leader bottleneck of Paxos. However, Lamport does not discuss in detail how to handle failures or how to prevent a slow leader from affecting others in a multi-leader setting. The idea of rotating the leader has also been used for a single consensus instance in the $\Diamond S$ protocol of Chandra and Toueg [10].

A number of low latency protocols have been proposed in the literature to solve atomic broadcast, a problem equivalent to the one of implementing a replicated state machine [10]. For example, Zieliński presents an optimistic generic broadcast protocol that allows commutable messages to be delivered in any order and requires $n > 3f$ [35], and elsewhere presents a protocol that relies on synchronized clocks to deliver messages in two communication steps [36]. Similar to Mencius, the latter protocol sends *empty* (equivalent to *no-op*) messages when it has no message to send. Unlike Mencius, it suffers from higher latency after one server has failed. The Bias Algorithm minimizes delivery latency when the rates at which processes send messages are known in advance [2], an assumption that Mencius does not make. Schmidt *et al.* propose the M-Consensus problem for low latency atomic broadcast and solved it with Collision-fast Paxos [32]. Instead of learning a single value for each consensus instance, M-Consensus learns a vector of values. Collision-fast Paxos works similar to Mencius as it requires a server to propose an empty value when it has no value to propose but differs in its way of handling failures: it allows a non-faulty server to take over the slot of a faulty server, which makes out-order-commit impossible when Collision-fast Paxos is used to implement atomic broadcast.

We are not the first to consider high-throughput consensus and fault-scalability. For example, FSR [16] is a protocol for high-throughput total-order broadcast for clusters that uses both a fixed sequencer and ring topology. PBFT [7] and Zyzzyva [19] propose practical protocols for high-throughput consensus when processes can fail arbitrarily. Q/U [1] proposes a scalable Byzantine fault-tolerant protocol.

Steward [3] is a hybrid Byzantine fault-tolerant protocol for multi-site systems. It runs an Byzantine fault-tolerant protocol within a site and benign consensus protocol in between sites. Steward could benefit from Mencius by replacing their inter-site protocol (the main bottleneck of the system) with Mencius.

9 Future work and open issues

The following are issues that require further work. In the interest of space, we only mention them briefly.

Byzantine failures It is not straightforward to derive a "Byzantine Mencius", because skipping, the core technique that makes Mencius efficient, is not built on a quorum abstraction. We plan to explore a Byzantine version of Mencius by applying techniques such as Attested Append-only Memory [11].

Coordinator allocation Mencius's commit latency is limited by the slowest server. A solution to this problem is to have coordinators at only the fastest $f + 1$ servers and have the slower f servers forward their requests to the other sites.

Sites with faulty servers We have assumed that while a server is crashed, it is acceptable that its clients do not make progress. In practice, we can relax this assumption and cope with faulty servers in two ways: (1) have the clients forward their requests to other sites, or (2) replicate the service within a site such that the servers can continuously provide service despite the failure of a minority of the servers.

10 Conclusion

We have derived, implemented, and evaluated Mencius, a high performance state machine replication protocol in which clients and servers are spread across a wide-area network. By using a rotating coordinator scheme, Mencius is able to sustain higher throughput than Paxos, both when the system is network-bound and when it is CPU-bound. Mencius presents better scalability with more servers compared to Paxos, which is an important attribute for wide-area applications. Finally, the state machine commit latency of Mencius is usually no worse,

and often much better, than that of Paxos, although the effect of network variance on both protocols is complex.

Acknowledgements This material is based upon work supported by the National Science Foundation under Grant No. 0546686. We would like to thank our shepherd Hakim Weatherspoon for helping us prepare the final version of the paper, the anonymous reviewers for their helpful and insightful comments, and the DETER testbed for the experimental environment.

References

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, et al. Fault-scalable Byzantine fault-tolerant services. *SIGOPS Oper. Syst. Rev.*, 39(5):59–74, 2005.

[2] M. Aguilera and R. Strom. Efficient atomic broadcast using deterministic merge. In *Proceedings of ACM PODC*, pages 209–218, New York, NY, USA, 2000.

[3] Y. Amir, C. Danilov, J. Kirsch, et al. Scaling Byzantine fault-tolerant replication to wide area networks. In *Proceedings of IEEE/IFIP DSN*, pages 105–114, Washington, DC, USA, 2006.

[4] Y. Amir, L. Moser, P. Melliar-Smith, et al. The Totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst.*, 13(4):311–342, 1995.

[5] T. Benzel, R. Braden, D. Kim, et al. Design, deployment, and use of the DETER testbed. In *Proceedings of the DETER Community Workshop on Cyber-Security and Test.*, Aug 2007.

[6] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of OSDI*, pages 335–350, Berkeley, CA, USA, 2006.

[7] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, Nov. 2002.

[8] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of ACM PODC*, pages 398–407, 2007.

[9] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[10] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

[11] B. Chun, P. Maniatis, S. Shenker, et al. Attested append-only memory: making adversaries stick to their word. In *SOSP*, pages 189–204, 2007.

[12] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[13] D. Dobre, M. Majuntke, and N. Suri. CoReFP: Contention-resistant Fast Paxos for WANs. Technical Report TR-TUD-DEEDS-11-01-2006, Department of Computer Science, Technische Universität Darmstadt, 2006.

[14] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. In *Proceedings of ACM PODS*, pages 1–7, New York, NY, USA, 1983.

[15] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[16] R. Guerraoui, R. Levy, B. Pochon, et al. High throughput total order broadcast for cluster environments. In *Proceedings of DSN*, pages 549–557, Washington, DC, USA, 2006.

[17] S. Hemminger. Network emulation with NetEm. In *Linux Conf Au*, April 2005.

[18] F. Junqueira, Y. Mao, and K. Marzullo. Classic Paxos vs. Fast Paxos: Caveat emptor. In *Proceedings of the 3rd USENIX/IEEE/IFIP Workshop on Hot Topics in System Dependability (HotDep'07)*, 2007.

[19] R. Kotla, L. Alvisi, M. Dahlin, et al. Zyzzyva: Speculative Byzantine fault tolerance. In *SOSP*, pages 45–58, 2007.

[20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[21] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[22] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[23] L. Lamport. Lower bounds on asynchronous consensus. In *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 22–23, 2003.

[24] L. Lamport. Generalized Consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[25] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, Oct. 2006.

[26] L. Lamport, A. Hydrie, and D. Achlioptas. Multi-leader distributed system. U.S. patent 7,260,611 B2, Aug 2007.

[27] J. Lorch, A. Adya, J. Bolosky, et al. The SMART way to migrate replicated stateful services. In *Proceedings of the ACM SIGOPS EuroSys*, pages 103–115, New York, NY, USA, 2006.

[28] J. MacCormick, N. Murphy, M. Najork, et al. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of OSDI*, pages 105–120, Berkeley, CA, USA, 2004.

[29] Y. Mao, F. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for WANs. Technical Report CS2008-0930, Dept. of Computer Science and Engineering, UC San Deigo, 2008.

[30] J. Nagle. RFC 896: Congestion control in IP/TCP internetworks, Jan. 1984.

[31] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev.*, 27(1):31–41, 1997.

[32] R. Schmidt, L. Camargos, and F. Pedone. On collision-fast atomic broadcast. Technical Report LABOS-REPORT-2007-001, Éole Polytechnique Fédérale de Lausanne, 2007.

[33] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, pages 299–319, Dec. 1990.

[34] J. Wensley, L. Lamport, J. Goldberg, et al. Sift: Design and analysis of a fault-tolerant computer for aircraft control. *Tutorial: hard real-time systems*, pages 560–575, 1989.

[35] P. Zieliński. Optimistic generic broadcast. In *Proceedings of the 19th International Symposium on Distributed Computing*, pages 369–383, Kraków, Poland, September 2005.

[36] P. Zieliński. Low-latency atomic broadcast in the presence of contention. *Distributed Computing*, 20(6):435–450, 2008.

[37] ZooKeeper. `http://hadoop.apache.org/zookeeper`.

Notes

[1] Mencius, or Meng Zi, was one of the principal philosophers during the Warring States Period. During the fourth century BC, Mencius worked on reform among the rulers of the area that is now China.

[2] There are other structures of state machines, such as a primary-backup structure where only one server executes the command and communicates the result to the rest of the servers, or one in which a command generates multiple responses, each sent to different clients. Our protocol can be adapted to such structures.

[3] To eliminate trivial implementations, we require that there exists an execution in which the coordinator proposes a value $v \neq$ *no-op* that is chosen as the consensus value.

[4] In practice, one non-faulty server is elected to lead the revocation process to avoid wasting resources or causing liveness problems. See [29] for more detailed discussion.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

For more information about membership and its benefits, conferences, or publications, see http://www.usenix.org.

SAGE, a USENIX Special Interest Group

SAGE is a Special Interest Group of the USENIX Association. Its goal is to serve the system administration community by:

- Establishing standards of professional excellence and recognizing those who attain them
- Promoting activities that advance the state of the art or the community
- Providing tools, information, and services to assist system administrators and their organizations
- Offering conferences and training to enhance the technical and managerial capabilities of members of the profession

Find out more about SAGE at http://www.sage.org.

Thanks to USENIX & SAGE Corporate Supporters

USENIX Patrons

Google

Microsoft Research

NetApp

USENIX Benefactors

hp invent

IBM

Linux Pro Magazine

vmware

USENIX & SAGE Partners

Ajava Systems, Inc.

DigiCert® SSL Certification

FOTO SEARCH Stock Footage and Stock Photography

Splunk

Zenoss

USENIX Partners

Cambridge Computer Services, Inc.

GroundWork Open Source Solutions

Hyperic

Infosys

Intel

Oracle

Sendmail, Inc.

Sun Microsystems, Inc.

Xirrus

SAGE Partner

MSB Associates

ISBN-13: 978-1-931971-65-2